AFRL-RI-RS-TR-2014-111

USING SOFTWARE GENERATION AND REPAIR FOR CYBER-DEFENSE

KESTREL INSTITUTE

MAY 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-111 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
PATRICK M. HURLEY
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 2014	FINAL TECHNICAL REPORT	SEP 2010 – SEP 2013

4. TITLE AND SUBTITLE

USING SOFTWARE GENERATION AND REPAIR FOR CYBER-DEFENSE

5a. CONTRACT NUMBER
FA8750-11-C-0005

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F & 63788F

6. AUTHOR(S)

Stephen Fitzpatrick, Cordell Green, Stephen Westfold, and James McDonald

5d. PROJECT NUMBER
G2MG

5e. TASK NUMBER
R0

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2014-111

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2014-1876
Date Cleared: 21 April 2014

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This work investigates the use of medium-grained synthetic diversity to inhibit an attacker's ability to identify and exploit weaknesses in a networked application. We specify the application and use formal refinements to generate numerous implementations that use different combinations of algorithms and data representations. By deploying different implementations we make it more difficult for an attacker to learn details about how any particular implementation works, what resources it uses and what operating system or network services it uses – such information is a critical prerequisite for many cyber attacks. We also use semantic constraints in specifications and refinements to generate run-time monitors that can detect compromised data. The semantic information also enables either total or approximate repair of the data, allowing an application to recover from an attack. We also use formal transformations on specifications to augment data structures with additional semantic constraints to enhance monitoring and repair.

15. SUBJECT TERMS

Formal methods, formal specification, program refinement, code generation, synthetic diversity, moving target defense, cyber attacks.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**PATRICK M. HURLEY**
U	U	U	UU	65	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

List of Figures

List of Tables

1 SUMMARY

The research reported here has the following objectives:

- to inhibit an attacker's ability to identify weaknesses in an application;
- to limit an attacker's ability to exploit an identified weakness;
- to detect data that has been compromised by an attack;
- to repair compromised data.

The overall goal is to enhance an application's ability to survive attacks during deployment, and thus to allow the system of which it is a component to continue operating (perhaps at a reduced level of performance).

The research builds on technology from the field of automated software generation, in particular:

- specification technology allows the formal statement of: (i) the objectives of an application, and (ii) design and implementation techniques;
- automated refinement technology allows executable code to be generated from a specification.

These technologies are used as follows:

- **Synthetic diversity**: For any given application specification, there are numerous ways to generate executable code, using alternative algorithms, alternative data type representations, alternative run-time libraries, etc. Varying the generated code varies the details of how the application represents its data, how it performs its computations, what resources it needs, how it interacts with other software or network services, etc. Knowledge of such details is critical for many forms of attack – varying them thus makes it harder for an attacker to launch and sustain a successful attack.
- **Run-time monitoring and repair**: Specifications and refinements contain rich semantic information that constrain data representations and computations. Run-time monitors are generated from such constraints. If an attack causes an application to violate these constraints, the violation may be detected by the monitors, which initiate corrective action. The semantic constraints can also be used to generate repair mechanisms that restore semantic consistency to data that has been compromised by an attack – the repair may be partial or full, depending on how strong the constraints are and the extent of the damage to the data.

In addition, the research seeks to understand the role that synthetic diversity can play in cyber defense.

2 INTRODUCTION

A software application may be subject to various forms of cyber attack. An attack may originate from any network to which the application's host computer/system is connected, either directly in the form of malicious inputs sent to the application or indirectly in the form, say, of a denial of service attack that deprives the application of needed resources (such as memory, CPU time, network bandwidth or services running on remote hosts).

Alternatively, an attack may originate from malware running on the application's host. The malware may have been implanted offline (e.g., from a storage device) or it may have been injected by a network attacker exploiting a weakness in another application or service. Such co-resident malware may attack an application by tampering with shared data (e.g., data stored in a file system), by consuming excessive resources or by monitoring the application for information leaked via side channels (manifest, e.g., in the application's CPU or network usage patterns).

Many forms of attack require the attacker to acquire detailed knowledge about the application, and then specifically craft the attack according to those details. For example:

- If an attacker wishes to cause an application to behave in a specific way, then the attacker may need to know how the application's data is represented in the file system.
- If an attacker wishes to cause an application to stall, then knowledge about which remote services the application depends on may allow an attack to be carried out stealthily, by cutting off access to just those services.
- If an attacker knows that an application uses an algorithm that needs a lot of memory, then the attacker can more effectively attack the application using a denial of service attack that depletes the available memory, rather than one that depletes the available computational power.
- Side channel attacks depend on correlations between visible signals and hidden data or processes.

To acquire such knowledge, an attacker may need to surveil the application or analyze its code, either of which may be time-consuming.

In this introduction, we give a brief overview of some areas of technology that aim to thwart cyber attacks:

- *synthetic diversity*, in which many versions of an application are deployed, with the aim of restricting an attacker's ability to exploit knowledge learned through surveillance or analysis;
- *run-time monitoring*, in which the behavior of an application is checked against expectations, with the aim of detecting behavior that arises from a compromise;
- *run-time repair*, in which data redundancy or constraints are used to partially or completely restore compromised data.

We also give an introduction to the Specware tool, which we use: to specify data structures, algorithms and libraries; and to develop software refinements and transformations that are the basic tools that we use to implement synthetic diversity, run-time monitoring and run-time repair.

2.1 Synthetic Diversity

Synthetic diversity is a defensive technique based on causing an application to vary the details of how it works or how it represents its data, quickly enough to render unusable any knowledge that an attacker might acquire. Synthetic diversity does not seek to make absolute guarantees about an application's defense; rather, it seeks to raise the cost of a successful attack. It is complementary to other defense techniques such as virus scanning, intruder detection and code shepherding.

Prior research on synthetic diversity has typically focused on fine-grained aspects of an application's code. For example:

- With instruction set randomization [1,2], a virtual processor is created that is essentially the same as some real processor, but with its op-codes randomly permuted. Valid code is transformed to reflect the op-code permutation, prior to loading into the virtual processor – it is thus executed essentially the same as the original code would be on the real processor. However, if an attacker manages to inject code into an application running on the virtual processor, the injected code (presumably) uses the original op-codes rather than the permuted op-codes – the injected code is thus essentially gibberish and its execution by the virtual processor will likely soon result in a crash (which is considered a better outcome than the injected code subverting the application).
- With address space randomization [3], the positions of data in memory is not (likely) predictable by an attacker. An attacker trying to modify some datum does not know its address; an attacker who succeeds in overwriting a data area (e.g., the stack) with code does not know the injected code's address, which makes it harder to cause the processor to begin executing the injected code.

This research addresses larger-grained aspects of synthetic diversity, such as how data structures are represented and which algorithms are used. For example, an application may represent a Set (over a bounded enumeration) using either a List (containing each element of the Set) or a Bit Vector (with the i^{th} bit 1 iff the i^{th} element of the enumeration is in the Set). If an attacker creates an attack that tries to change the Set and expects the former representation, the attack will almost certainly fail if the latter representation is in use.

Likewise, different algorithms may be functionally equivalent but have different resource usage patterns. If an attacker knows which algorithm an application uses, the attacker may be able to selectively target the particular resources that are critical to that algorithm. For example, a denial of service attack on a CPU-intensive algorithm would be different from a denial of service attack on a memory-intensive algorithm.

In addition, certain algorithms have extreme worst-case resource usage. For example, some regular expression matching algorithms can consume excessive amounts of CPU time trying to process certain regular expressions. If an attacker knows that an application uses such an algorithm, the attacker may launch an *algorithmic complexity attack*, in which specifically crafted input triggers the worst-case behavior.

Moreover, knowing which algorithms are used by an application helps in analyzing side channels.

As a final example, a weakness in an application may be caused by a bug in a routine imported by the application from an external library. Knowing which libraries an application uses may allow an attacker to deliberately trigger the bug. Varying the libraries makes this more difficult.

Table 1 shows several granularities of synthetic diversity along with the potential effects they have on attacks. This research is primarily focused on diversity of algorithms and data structures, with some development of obfuscation techniques. The technologies developed also allow for diversity of code libraries and message serialization randomization, but these were not explored. The technologies should also be of use in diversifying protocols and architectures, but additional technologies or semantic theories may be needed.

Table 1 Granularity of synthetic diversity

Granularity of Synthetic Diversity	Effect
Architecture	fundamentally different patterns of interaction
Protocol	fundamentally different behaviors
Algorithm	fundamentally different patterns of resource use
Data structure	fundamentally different constraints on data; fundamentally different organization of data
Obfuscation techniques	source/binary harder to analyze
Code libraries	different underlying flaws & resource use
Message serialization randomization	messages harder to spoof
Address space randomization	attack overwrites wrong locations; injected code cannot be invoked
Instruction set randomization	injected code fails due to wrong op codes

These different granularities of synthetic diversity are complementary. In particular, since the technologies developed in this research generate code in standard programming languages, it should be possible to use whatever technologies exist to perform address space and instruction set randomization.

2.2 Run-time Monitoring

The aim of run-time monitor is to detect when an application is behaving in ways that it should not, due to faults in the software or to the actions of an attacker. There are various forms of run-time monitoring of applications.

- With permissions-based monitoring [4], an application is designated as being allowed to perform certain classes of operation or access certain classes of resources. For example, an "app" on a mobile phone may be granted permission to access the phone's calendar, but not its address book. The operating system prevents the app from going outside its permissions.
- With program shepherding [5], the run-time call stack is used to mediate access to resources and operations. This can be used to make sure that critical routines are invoked only through expected channels, rather than from, say, injected code.
- With taint tracking [6], data manipulated by an application is tagged as being trusted or untrusted, say. When trusted data is combined with untrusted data, the result is untrusted; in other cases, the result has the same status as the input data. Critical operations can be augmented with guards that apply sanity checks to any inputs that are untrusted (in particular). Taint tracking can help to prevent attacks such as SQL injection or shell injection since the injection relies on an attacker's ability to insert characters that have special significance to the back-end system (the SQL database or the shell interpreter) – since the attacker's data is untrusted, such characters can be eliminated by the guards.
- With code profiling, training data is used to generate statistical profiles of how an application behaves; e.g., how much CPU time it typically requires or which parts of the file

system it accesses. The profile can be used to restrict the application's behavior during deployment.

These technologies are useful in inhibiting attacks. One commonality they have, though, is that they attempt to characterize an application *post hoc*, either through analysis, observation or manual attribution.

In contrast, in this research, formal specifications are used to precisely capture the semantics of applications, data structures and algorithms; run-time monitors are synthesized from the semantics. A simple example is the specification of a function to sort a sequence: the output of the function must be sorted (according to some ordering relation) and must be a permutation of the input. Various monitors can be synthesized from this specification, exhibiting different trade-offs between thoroughness and cost:

- Rigorously checking that the output is a sorted permutation of the input is possible but perhaps too expensive.
- Checking that the output is sorted can be performed with a linear scan of the output. Instead of checking that the output is a permutation of the input, a monitor might only check that they have the same length, say.
- Rather than check that the entire output is sorted, a monitor might only sample various positions along the Sequence.
- Rather than check every output of the function, a monitor might check only randomly selected outputs.

In general, if the result, *R*, must satisfy some constraint *p(R)*, then a monitor may check any weakening of *p* (any necessary constraint); i.e., any *q(R)* such that $p(R) \Rightarrow q(R)$. The weakest constraint is *q(R)=true*, which requires no run-time checking at all.

Of course, any combination of such monitors is also possible – one might be selected at random for each invocation of the function.

Any constraint that involves unbounded quantification cannot be checked at run-time; e.g., a constraint over all sequences or over all integers. Bounded quantification may be checkable, if the technology can recognize the bounds.

In this research, a run-time monitor checks that semantic constraints are being observed. Violations may arise from several sources:

- Data obtained from a library routine may not meet its specification, perhaps because of a bug or because the routine has been compromised by an attack.
- Likewise, data shared with another application or service may be corrupted.
- There may be errors in the generated code. This should be unlikely, but it cannot be completely ruled out until all parts of the generation process have been rigorously proved (a requirement that is being pursued in related research).

2.3 Run-time Repair

Run-time repair of data addresses two problems: when information may be lost because an attacker has deleted data; when an application may malfunction because an attacker has corrupted data (e.g., the application may crash or hang). In the former case, the objective is to

restore the data. In the latter case, the objective is to put the data into a structurally and semantically sound state so that the application can continue to operate.

In addition to well-known methods of repairing data using error-correcting codes or redundant data structures, there are also several novel repair technologies being developed:

- With failure oblivious computing, an operation that is detected to be erroneous is replaced with a similar, sound operation. For example, an attempt to read a sequence at an index that is out of bounds might be replaced with a read at any index that is in bounds. There is no guarantee that such replacements are correct, but they prevent dangerous operations (particularly when data is being written) and, for some classes of application, appear to allow the application to continue operating.
- With taint tracking, as discussed above, untrusted data that is found to contain special elements may be sanitized before being passed to a critical operation. For example, an SQL query may contain unescaped quote marks that are untrusted – these may be escaped before the query is processed, preventing injection attacks.

For the research reported here, repairs can be formalized as follows: given a data structure that violates its semantic constraints, find a data structure that satisfies its constraints and is as close to the original data structure as possible (according to some cost or distance metric). For example, consider a sequence of integers that is supposed to be sorted (according to the standard less-than order). If the sequence is found to contain some elements that are out of order, several repairs are possible: the elements might be moved into their correct positions in the sequence; the elements might be removed from the sequence; the elements might be altered to values that are in order.

None of these repairs is guaranteed to be correct, in the sense of restoring a compromised sequence back to its original, uncompromised state, unless there is sufficient knowledge about how many and what sort of alterations a sequence might be subjected to.

Nonetheless, in the spirit of "fighting through an attack", the application is likely to function better with the repaired data than with the compromised data – many of the application's operations presumably rely on the sequence being sorted and will misbehave if it is not. For example, a search algorithm might be optimized based on the assumption that the sequence is sorted. If such an algorithm encounters an out-of-place element, it may prematurely terminate, believing it has exhausted that part of the sequence that might contain the sought-after element. While the repaired sequence may contain a few incorrect elements, the search operation will be able to perform unhindered.

A repair may even be chosen at random from several possible repairs, in the absence of cost metrics.

Such technologies as error-correcting codes, redundant storage and voting schemes can be incorporated into the approach developed in this research.

2.4 Specware

Kestrel's Specware tool provides a high-level language for specifying data structures, algorithms and applications, without undue regard to their representation or implementation. It also provides meta-programming facilities that allow software to be manipulated (e.g., for many

implementations of an application to be generated). In this section, we introduce some of the main constructs.

2.4.1 Specifications and Semantic Constraints

The research reported here is based on the Specware system [7]. Specware supports the specification of software and the automated generation of executable code. Its specification language is similar to those in proof systems such as Isabelle/HOL [8] and Coq [9]. It allows data types to be defined and semantically constrained. For example, a data type for prime numbers may be specified as:

```
type Prime = Int | prime?
```

where the notation "type | predicate" indicates some pre-existing type (integer) that is constrained to satisfy the given predicate ("prime?", whose definition is not shown). It is a convention in use of Specware to give predicates a name that ends in a question mark, but that is not a requirement.

Such a type can be used to define other types, to constrain inputs to functions or to constrain the results of functions. For example, consider the following.

```
op factors(i:Int | i > 1): Sequence Prime
```

This declares a function (an "operation" or "op") called "factors" that takes as input an Integer, i, that is greater than 1 and returns a sequence of integers, each of which is prime.

For each invocation of a function, there is an associated *proof obligation* that requires that the inputs to the function satisfy their constraints – formally, a specification is not known to be valid until all of the proof obligations have been *discharged*. For example, *factors(-1)* is formally a type error in Specware.

Simple proof obligations can be automatically discharged by Specware. More complicated proof obligations may require the use of a theorem prover. For this research, though, the semantic constraints in the types are taken as candidates for run-time monitoring.

In normal development, the result type of a function is something that can be assumed to be true. Thus, any element of any sequence returned by a call to *factors* can be assumed to be prime. Such assumptions form part of the knowledge that allows the proof obligations for other function invocations to be discharged.

However, for this research, the semantic constraints in result types are taken as candidates for run-time monitoring.

2.4.2 Refinement and Semantic Constraints

In addition to declaring the type of a function, Specware allows the semantics of the function to be captured. For example, the *factors* function can be fully characterized by requiring that the sequence of primes is sorted and that the product of the primes equals the input to the function:

```
op factors(i:Int | i > 1): SortedSequence Prime =
  the(S) product(S) = i
```

The notation *the(x) p(x)* denotes the unique value that satisfies the predicate *p*, so this states that the function returns the unique sequence whose product is i.

The sortedness of the result is implied by the result type. It could alternatively have been incorporated into the *the* clause:

```
op factors(i:Int | i > 1): Sequence Prime =
  the(S) sorted?(S) && product(S) = i
```

Likewise, the result type implies that each element of the result sequence is prime.

A function specified using *the* or existential quantification – *ex(x) p(x)* – or universal quantification – *fa(x) p(x)* – is not directly executable. The function may have been defined only for specification purposes and may never need to be executed. However, if it is to be executed, then it must be *refined* into executable form. For example, a perhaps naïve implementation of *factors* is:

```
refine def factors(i:Int | i > 1): SortedSequence Prime =
  while((2, i, empty),
    fn (prime, n, factors) → n > 1,
    fn (prime, n, factors) →
      if prime = n || prime divides? n
      then (prime, n/prime, factors <| prime)
      else (nextPrime(prime), n, factors),
    fn (prime, n, factors) → factors)
```

This definition shows several aspects of Specware's language:

- An expression of the form *while(initial, continue?, next, final)* is a while loop. The arguments are: an initial value, a function that determines if the loop should continue, a function that gives the next value, and a function that, once the loop has terminated, gives the final value. In this example, the loop deals with a value having three components: a prime number (initially 2), the current value being factored (initially i), and a list of factors found so far (initially empty).
- An expression of the form *fn(x)* → *e* denotes an anonymous function with formal parameter *x* and body *e*.
- The expression *S* <| *e* denotes an element *e* appended to a sequence *S*.

(Other aspects of the Specware language will be introduced as needed.)

Because this definition of *factors* is a refinement of the specification of *factors*, the definition must produce a value that satisfies the specification. i.e., the value returned by this definition must be a sorted sequence of primes such that their product equals the argument given to *factors*.

As with the semantic constraints implied by types, the semantic constraints implied by refinement are required to be discharged before a specification is known to be valid. For this research, however, the semantic constraints are taken as candidates for run-time monitoring.

Note that run-time monitoring cannot, in general, check that there is a *unique* value satisfying a constraint. i.e., given the constraint *the(x) p(x)*, run-time monitoring can check if some value satisfies *p*, but not that it is the *only* satisfying value.

2.4.3 Morphisms

Refinements can be packaged together to perform a coherent task. For example, a specification for Set as an abstract data type might declare the type

```
spec Set
type Set E
```

and give signatures (types) and semantics for the usual functions; e.g., the union function may be declared as:

```
op [E] \/(S1:Set E, S2:Set E) infixr 24: Set E =
  the(U) fa(e:E) e in? U <=> (e in? S1 || e in? S2)
```

In this case, the function name is \/ and it is declared to be an infix operator (with right association and priority 24). The notation [E] introduces a type variable to represent the type of the Set's elements – the Set data type is polymorphic.

A package of refinements that represents a Set as a List would give a definition for the data type representation

```
spec SetAsList
type Set E = List E | nonRepeating?
```

along with definitions for the functions; e.g.,

```
op [E] \/(L1:Set E, L2: Set E) infixr 24: Set E =
    let diff = filter (fn e → e nin? L2) L1
    in diff ++ L2
```

(where the operator *nin?* tests if an elements is *not* in a Set and ++ concatenates two Lists).

We formally connect the abstract Set to the more concrete SetAsList using a *morphism*:

```
SetAsListM = morphism Set -> SetAsList {}
```

This defines a morphism called "SetAsListM" that has the Set specification as its source and the SetAsList specification as its target. This morphism can be applied to a specification, with the effect of implementing the abstract data type (Set) in terms of a concrete data type (List). For example, if SomeServer is a specification for an application that uses the abstract Set data type, then SomeServer[SetAsListM] is a refined specification that instead uses the concrete data type List. (The braces after the morphism target allow for type or function renamings to be given. In this case, they are not needed.)

Roughly speaking, when used this way, specification/refinement is similar to features such as interfaces/implementations in languages such as Java. However, the critical difference is that refinements are required to preserve the semantics of the source specification – whatever can be proved about the SomeServer specification must also be provable about the refined specification SomeServer[SetAsListM].

Refinements can be chained together; e.g., SomeServer[SetAsListM][MapAsHashTableM] applies SetAsListM and then MapAsHashTableM. Since each refinement preserves the semantics of the source specification, their combination also preserves the semantics.

2.4.4 Transformations

In addition to refinement, specifications can be produced by *transformation*. Examples of commonly used transformations include:

- Unfolding, in which modular definitions are in-lined. The objective is to provide context for a computation to allow it to be simplified.
- Algebraic simplification, in which such rules as *head(cons(x,L)) = x* and *if true then e else f = e* are used to simplify computations.
- Common sub-expression elimination, in which a significant computation that appears twice in some expression is bound to a variable so that it is only computed once.

A single transformation typically has a well-defined objective. Complex changes can be achieved by chaining transformations. Typically, a transformation is required to preserve the semantics of the source specification.

Specware's notation for the application of a transformation is similar to its notation for applying a refinement: *S{T}* denotes the application of transformation *T* to specification *S*. Transformations and refinements can be intermingled: e.g.,

SomeServer[SetAsList][MapAsHashTable]{Simplify}{CommonSubexpressions}

This notation provides a compact way for denoting how a specific, concrete, optimized specification (and after code generation, a specific implementation) can be produced from an abstract specification. It thus provides an efficient means for expressing the production of many implementations from an original specification.

2.4.5 Code Generation

When a specification is as fully concrete as it needs to be (recalling that some functions may not be intended to be executable), Specware can generate executable code. For research, the target programming language is typically Lisp, but other languages such as C and Java can be targeted. This research focused mainly on Lisp.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

In this section, we detail our approach to synthetic diversity, run-time monitoring and run-time repair.

For synthetic diversity, our approach starts with high-level specifications of data structures and algorithms that are devoid of details concerning representation or implementation. Using such high-level specifications, we construct a specification of an application. This specification defines what we need of the application, but says little about how the application is to carry out its computations. This leaves us free to realize those computations in any manner consistent with the specification – i.e., we are free to generate diverse implementations.

The second part of our approach to synthetic diversity is to develop multiple representations of data structures (e.g., a Set can be represented as a List or as a Hash Table) and multiple implementations of algorithms (e.g., QuickSort, MergeSort and BubbleSort are all implementations of sorting).

We then *mix and match* the representations/implementations to realize the high-level data structures and algorithms used in the specification of the application. We use automated tools to generate all possible (valid) combinations.

Some standard transformation techniques can be applied to the resulting code to further increase diversity – we consider one such technique as an example.

We also consider how to measure the diversity of a collection of implementations, and how to ensure that we are generating diverse collections.

For run-time monitoring, our approach starts with the rich semantic information present in specifications (data type invariants and function pre- and post-conditions) and in the relationship between a high-level specification and a lower-level implementation. We use this semantic information to ensure an application behaves correctly, by generating run-time monitors.

For run-time repair, our approach is based on data type invariants – these are constraints on a data structure and when a run-time monitor detects a failure of a constraint, it applies a repair operator to restore the constraint.

We also consider several methods for adding constraints to data structures (e.g., by adding hash codes or redundancy).

3.1 Multiple Data Representations

For this research, synthetic diversity is focused primarily at the granularity of data structures and algorithms. So, we will first show how Specware can be used to allow multiple data representations to be used. We will use as a simple example complex numbers, which have two common representations: Cartesian, in which the real and imaginary parts are stored; polar, in which the radius (magnitude) and argument (angle) are stored.

3.1.1 Representation-free Specification

In order to use either, or both, of these representations, we first define a specification, Complex, that captures the commonality between the representations.

```
Complex = spec
type Complex

op real(x: Complex): Real
op imag(x: Complex): Real
op rad(x: Complex): NonnegReal
op arg(x: Complex): Angle
```

We declare a type Complex. Since no definition is given, we are not yet committing to any specific representation.

We declare four *observer* functions on values of type Complex, to obtain the real and imaginary parts, and the radius and argument. The latter two are constrained to return, respectively, a real number greater than or equal to 0, and an Angle, i.e., a real number in the range 0 (inclusive) to 2π (exclusive). (In this report, we will gloss over the difference between real numbers and their finite representation as floating point values.)

While the observers real & imag might be naturally associated with a Cartesian representation, and rad & arg with a polar representation, all four observers are compatible with any representation for Complex that we choose.

We next mutually constrain these functions by adding an axiom that states when two complex numbers are equal.

```
axiom complex_equality is
  fa(x: Complex, y: Complex)
        (x = y <=> real x = real y && imag x = imag y)
    && (x = y <=>  rad x = rad y  &&  arg x = arg y)
```

In effect, this axiom states that the functions real and imag form a complete set of observers for the Complex type – everything that can be known about a Complex, can be found through these functions. Likewise, the functions rad and arg also form a complete set of observers.

We now introduce functions for constructing values of type Complex. Since real & imag and rad & arg both form complete sets of observers, we define a construction function for each.

```
op mkCartesian(r: Real, i: Real): Complex =
  the (x) real x = r && imag x = i


op mkPolar(r: NonnegReal, t: Angle | uniqueZero?(r, t)): Complex =
  the (x) rad x = r && arg x = t
```

The mkCartesian function takes two real values and returns the unique Complex value that has the given values as the result of applying the real and imag observers.

The mkPolar function takes a non-negative real number and an Angle and returns the unique Complex value that has the given values as the result of applying the rad and arg observers. This function has an additional constraint on its arguments: if the radius is 0, then the argument must also be 0:

```
op uniqueZero?(r:NonnegReal, t:Angle): Bool =
  r = zero => t = zero
```

Since the argument of a complex number is essentially undefined when the radius is 0, forcing the argument to be 0 simplifies some technical details.

Note that we still have not committed to any particular representation. The two constructors, mkCartesian and mkPolar, are well defined regardless of which representation we use.

At this point of the development, we have two ways to construct any Complex value that we need, and two sets of observers for querying any Complex value. We can use these functions to develop a comprehensive set of functions for complex arithmetic. For example:

```
op *(x: Complex, y: Complex) infixl 27: Complex =
  mkCartesian(real x * real y - imag x * imag y, real x * imag y + imag x * real y)

op /(x: Complex, y: Complex | rad y > zero) infixl 26: Complex =
  mkPolar(rad x / rad y, Angle.subtract(arg x, arg y))
```

Here, we choose to use the mkCartesian function to define complex multiplication and the mkPolar function to define complex division. These choices are a matter of convenience and clarity and do not commit us to any particular representation. Note that the Angle.subtract function ensures that its result, the difference between two Angles, lies in the interval $[0, 2\pi)$.

In this way, we can develop a specification for Complex without committing to any particular representation.

3.1.2 Introducing Representations

Having developed a representation-free specification, we next develop specifications for particular representations. For any representation of Complex, we need to show how the observers and constructors relate to the representation. For a Cartesian representation:

```
Cartesian = spec
import Complex
type Complex = {real: Real, imag: Real}

def real x = x.real
def imag x = x.imag

def rad x = sqrt(sq(real x) + sq(imag x))
def arg x = angle(real x, imag x)

refine def mkCartesian(x, y) = {real = x, imag = y}
refine def mkPolar(r, t) = mkCartesian(r * cos t, r * sin t)

end-spec
```

This specification defines the type Complex to have two Real fields, called real and imag. The real and imag observer functions simply return the values of these fields. The rad and arg observers perform some trigonometry calculations to produce the appropriate values based on these fields.

Likewise, the mkCartesian constructor simply records its arguments in the fields, and the mkPolar constructor calculates the appropriate values for the fields from its arguments.

This specification contains all that we need to add to the Complex specification in order to have a concrete representation. We formally link the abstract Complex specification with this representational specification using a morphism:

```
CartesianM = morphism Complex -> Cartesian {}
```

Likewise, the minimal specification for the polar representation is:

```
Polar = spec
import Complex
type Complex = {rad: NonnegReal, arg: Angle} | uniqueZero?(rad, arg)

def real x = rad x * cos(arg x)
def imag x = rad x * sin(arg x)

def rad x = x.rad
def arg x = x.arg

refine def mkCartesian(x, y) = mkPolar(sqrt(sq x + sq y), angle(x,y))
refine def mkPolar(r, t) = {rad = r, arg = t}
```

This defines a representation for Complex in terms of two fields, one for the radius and one for the argument. The fields are constrained by their types and the requirement for a unique zero. The rad & arg observers and the mkPolar constructor directly access the fields, while the real & imag observers and the mkCartesian constructor perform the appropriate calculations for the fields.

We also have the option of giving alternative definitions of functions defined in the Complex specification. For example, we may wish to use the following definition of complex multiplication when the polar representation is used:

```
refine def *(x: Complex, y: Complex): Complex =
  mkPolar(rad x * rad y, addAngle(arg x, arg y))
```

This definition is, of course, functionally equivalent to the original definition, but is more efficient for the polar representation. For the purposes of synthetic diversity, it is useful to vary the function definitions between representations.

(Strictly speaking, the new definition is not a refinement of the original definition since they are functionally equivalent, but the same notation is used in Specware.)

We also define a morphism for the polar representation:

```
PolarM = morphism Complex -> Polar {}
```

Having defined the two morphisms, we can choose how to represent complex numbers. Suppose we have an application that uses complex numbers:

```
App = spec
import Complex
…
```

Then App[CartesianM] is a refined specification of the application in which complex numbers use the Cartesian representation and App[PolarM] is a refined specification in which complex numbers use the Polar representation.

This is the basic mechanism by which we construct implementations of an application diversified over data representations. Of course, different representations of a data structure tend to perform the operations in different ways – e.g., complex multiplication as performed by the Cartesian

representation and the polar representation. Thus, to an extent, diversification over data structures also gives a certain degree of diversification over algorithms. More algorithmic diversification will be discussed below.

3.2 Some Common Data Structures

Complex numbers are a simple data structure. We will next show how the approach to multiple representations works with more intricate data structures by considering some common data structures.

3.2.1 Finite Set

```
FiniteSet = spec
type FiniteSet E
axiom FiniteSet_equality is [E]
  fa(S1:FiniteSet E, S2:FiniteSet E) S1 = S2 <=> (fa(e:E) e in? S1 <=> e in? S2)
op [E] in? infixl 20 : E * FiniteSet E -> Bool
op [E] empty(): FiniteSet E = the(S) fa(e:E) ~(e in? S)
op [E] |>(e:E, S:FiniteSet E) infixr 25: FiniteSet E =
  the(S2) fa(e2:E) e2 in? S2 <=> (e2 in? S || e2 = e)
```

For finite sets, the primary functions that need to be addressed are:

- Membership: x in? S tests if the value x is a member of the set S.
- The empty set: empty() returns the unique set (of the given element type) that contains no elements.
- Insertion: x |> S adds the value x to the set S (if it is not already in S).
- Equality: S1 and S2 are equal iff they contain the same elements.

Given these functions, we can define all the common set functions. For example, set union can be defined as:

```
op [E] \/(S1:FiniteSet E, S2:FiniteSet E) infixr 24: FiniteSet E =
  the(U) fa(e:E) e in? U <=> (e in? S1 || e in? S2)
```

One function that is perhaps less straightforward to define without referencing a particular representation is set fold. A fold operation combines all of the values in a set into a single value by the repeated application of some binary function, the *folding* function – e.g., the sum of a set of integers can be obtained by folding with the integer + function.

The signature of fold is:

```
op [E, A] fold(f:A*E->A, init:A, S:FiniteSet E | foldable?(f)): A
```

where E is the type of the elements of the set and A is the type of the result. Note that these do not have to be the same. For example, to compute the set of words that appear in a set of strings:

```
fold(fn (words:Set(String), s:String) -> words \/ wordsIn(s), empty(), strings)
```

Here, E is String but A is Set(String).

The init parameter is the value with which the fold is initialized: 0 for summing, the empty set for finding the set of all words.

Different representations of a set may apply the folding function to the set's elements in different orders. Since the value returned by fold must not depend on which representation we use, we must constrain the folding function to not care about the order, as indicated by the foldable?(f) constraint in the signature.

Requiring the folding function to be associative and commutative would be sufficient, but it is too stringent – e.g., it would not allow the example above in which we compute the set of words in a set of strings with a fold. A less stringent but still sufficient constraint is:

```
op [E, A] foldable?(f:A*E->A): Bool =
  fa(a:A, e1:E, e2:E) f(f(a, e1), e2) = f(f(a, e2), e1)
```

This captures the essential condition that the order in which the elements are processed does not affect the overall result.

We can fully define fold without recourse to a representation, as follows:

```
op [E, A] fold(f:A*E->A, init:A, S:FiniteSet E | foldable?(f)): A =
  the(r:A)
    (empty?(S) => fold(f, init, S) = init) &&
    (fa(S1:FiniteSet E, e:E)
      e nin? S1 && (S1 <| e) = S => fold(f, init, S) = f(fold(f, init, S1), e))
```

This is a standard inductive definition with base case the empty set and inductive case a non-empty set. For the latter case, since we know nothing about how the elements are stored or ordered in any representation of a set, we use the form

```
e nin? S1 && (S1 <| e) = S
```

to pick out any element e of the set. This resembles non-deterministic choice, but it does not in fact make the definition non-deterministic because the foldable? constraint ensures that the final value of the fold is independent of the order in which elements are chosen.

3.2.1.1 Representation as a List

One representation for a set is a non-repeating list in which the order of the elements does not matter. The non-repeating aspect is simply captured as a type constraint – the unique? function below. The latter aspect, that the order does not matter, is expressed in Specware using a *type quotient*:

```
type UnordList E = (List E | unique?)/ equiv?
```

The equiv? function is an equivalence relation, in this case on non-repeating lists. It captures the notion of when two lists are to be considered the same (for the purpose of representing sets).

```
op [E] equiv?(L1:List E, L2:List E): Bool =
  forall? (fn e -> e in? L2) L1 &&
  forall? (fn e -> e in? L1) L2
```

This states that two lists are equivalent iff every element in L1 is also in L2, and vice-versa. Thus, [1, 6, 3, 4], [6, 4, 3, 1], [4, 1, 3, 6], … are all considered to be equivalent.

An element of the quotient type UnordList can thus be considered to be an equivalence class, consisting of all (non-repeating) lists that are equivalent. In practical terms, an arbitrary member

of the equivalence class is used to represent the class, e.g. the list [1, 6, 3, 4] may represent the class of lists L such that equiv?(L, [1, 6, 3, 4]).

The equality operator has fixed semantics in Specware. When two values of the same quotient type are tested for equality, the test is performed via the equivalence relation used to define the quotient type.

In a similar manner, suppose f is a function on a quotient type and v is a value of that type, then Specware generates proof obligations that require that f is independent of which representative might be used for v. This follows from the requirement that $x = y \Rightarrow f(x) = f(y)$, since $x = x$ regardless of which representative is used on the left and which on the right.

For example, the size function on sets can clearly be implemented using the length function on lists, and the value returned is independent of which list represents a given set (under the constraint that the list be non-repeating).

This is expressed as follows:

```
op [E] size(S:UnordList E): Nat =
  choose[UnordList] (fn L -> length(L)) S
```

This can be read as:

- to compute the size of Set S,
- let L be any representative of the equivalence class representing the Set (choose[UnordList] (fn L -> …) S)
- and compute length(L).

The complementary aspect is when a set is computed. A single (non-repeating) list is computed as the representative, and then lifted to the equivalence class. For example:

```
op [E] empty():UnordList E =
  quotient[UnordList] []
```

Here, the empty Set is represented by lifting the empty List to the quotient type UnordList.

A typical pattern for implementing Set operations using UnordList is to get the representatives of the Set arguments using the choose operator, perform the appropriate List operations, and if necessary, lift the result back to the quotient type using the quotient operator. For example, the insert function is implemented as:

```
op [E] |>(e:E, S:UnordList E) infixr 25: UnordList E =
  choose[UnordList]
      (fn L -> if e in? L then S else quotient[UnordList] (e::L))
      S
```

We first obtain the representative L. We use the List operator in? to test if the element e is already in the list. If so, we return the original Set. Otherwise, we prepend the element to the representative and lift the result to the quotient type.

Note that this preserves the non-repeating constraint. Naturally, the correctness of the implementations of many of the Set functions depends on this constraint. An alternative representation of Sets as Lists allows repeated occurrences of an element, with an appropriate modification to the equivalence relation such that Lists are considered the same regardless of

whether they contain 1 or more occurrences of a given value. The implementations of the Set functions would need to be modified to take this into account.

Many of the types used in this research involve quotients. It is a crucial aspect in connecting abstract types and representations.

3.2.1.2 Representation as a Hash Set

An alternative representation of a FiniteSet is as a HashSet: the elements of the FiniteSet are partitioned into n *buckets* according to the elements' hash values, so that bucket i contains only elements e such that hash(e) mod n = i. This is straightforward to express in Specware:

```
op [E] hashed?(L:FinSeq(Bucket E)): Bool =
  fa(i:Nat) i < length(L) =>
    (fa(x:E) x in? L@i => hash(x) mod length(L) = i)

type HSBucketList E = FinSeq(Bucket E) | hashed?
```

FinSeq is a finite sequence (see Section 3.2.2). Each bucket is itself a set – for simplicity, we represent a bucket as an UnordList (Section 3.2.1.1).

Many FiniteSet operations are straightforward to implement given this representation. For example, to test if a FiniteSet is empty, we check if all of the buckets are empty:

```
op [E] empty?(H:HSBucketList E): Bool =
  forall?(H, fn B:Bucket E -> empty?(B))
```

However, one motivation in using a HashSet is to speed up such operations as membership testing: to determine if a value e is in a set, we need only check if it is in the appropriate bucket. So, to maintain good performance, the number of elements in any bucket should be kept small. This can be achieved by making the number of buckets, n, large. However, the larger n is, the more memory overhead is incurred; in addition, depending on the representation for finite sequence, a larger n may incur higher costs to access buckets (e.g., if the sequence is represented as a sequential list).

Thus, the sequence of buckets should expand and contract to maintain a balance between its own length and the sizes of the individual buckets. A simple scheme is to enlarge the sequence once the average bucket size surpasses some threshold, and to shrink the sequence once the average bucket size falls below some threshold.

To avoid repeated expansions and contractions as elements are added and removed during some computation, the former threshold should be larger than the latter. However, this means that the length of the sequence is not a function of the elements in the set; rather, it depends on the history of how elements where added and removed. In particular, multiple sequences may represent the same set.

Thus, we must form a quotient type over sequences of buckets, in which two sequences of buckets are equivalent iff they contain the same elements (regardless of how they are partitioned among the buckets).

type HashSet E = (HSBucketList E) / equiv?

A further consideration in implementing a FiniteSet as a HashSet arises from an optimization. If two sequences of buckets are the same length n, and both contain some element e, we know that e must be in the same bucket i in both, where i = hash(e) mod n. This allows many set operations to be optimized. For example, to form the union of two bucket sequences, each of length n, we can pairwise union the corresponding buckets; i.e., bucket i of the result is the union of bucket i of the first set and bucket i of the second. Note that the resulting sequence is automatically properly partitioned, since bucket i of the result contains only elements e satisfying the constraint hash(x) mod n = i.

3.2.1.3 Comparison of Representations

In general, we expect the HashSet representation to be more efficient than the UnordList representation for computations in which testing for membership dominates (directly or indirectly, e.g., in testing if one set is a subset of another). If, however, construction of Sets causes the expansion and contraction operations to be invoked often, then the UnordList representation may be more efficient.

An UnordList representation is constrained to be non-repeating. This is a constraint that can be monitored at run-time – however, it is quite expensive to check thoroughly.

Each bucket in a HashSet is an UnordList, and so is individually constrained to be non-repeating. Since it contains many fewer elements than the whole set, checking this constraint would not be as expensive.

In addition, each element in a bucket must satisfy the hashing constraint. This can be checked element by element, and so is well suited to sampling at run-time.

It should be noted, however, that hashing must be semantically consistent: if two values are semantically equal, they must have the same hash. This means that a value's hash cannot be derived from a non-semantic property such as an address in memory (unless some form of interning is performed, which forces semantically equal values to have the same address). Depending on how it is implemented, hashing may thus be expensive, although such expenses may be offset if the hash can be cached.

3.2.2 Finite Sequence

Essentially, a finite sequence of length n is a mapping from whole numbers in the range [0, n) to some element type. Thus, there are two primary observers: one to retrieve the length, and another to retrieve the i^{th} value.

```
FiniteSequence = spec
type FinSeq E
op [E] length(S:FinSeq E): Nat
op [E] @(S:FinSeq E, i:Nat | i < length(S)) infixl 30: E
```

The canonical constructor for a finite sequence is tabulate:

```
op [E] tabulate(n:Nat, f:Nat->E): FinSeq E =
  the(S:FinSeq E)
    length(S) = n &&
    fa(i) i < length(S) => S@i = f(i)
```

Given a length n and a function f, tabulate returns a finite sequence of length n and having element i equal to f(i).

Given these functions, it is straightforward to define two other common constructors (e.g., to construct the empty sequence and to insert and element at a given index) and the common sequence functions such as applying a given function to each element of a sequence (map), combining all of the elements into a single value (folding – an example is summing all of the elements), find the indices at which a given value occurs, and extracting a subsequence.

While we generally consider sequences to be based on random access to the elements and lists to be based on sequential access, we do not enforce a strict separation. For convenience, we can also define list-like functions for sequences: prepend an element, extract the first element, and remove the first element. In addition, we can represent a sequence as a list.

3.2.2.1 Representation as a List

The representation of a sequence as a list is straightforward: a sequence of length n is represented as a list of length n, and the i^{th} element of the sequence is the i^{th} element of the list. (Note that any permutation of the elements would be acceptable.)

One minor consideration is that many sequence functions deal with indices. For example, a folding function may take an element's index as well as its value as argument. In many cases, there is a standard analogous function on lists that takes just the value as an argument. This means that we cannot directly use many of the standard list functions. However, it is straightforward to provide alternative list functions that do deal with the indices.

3.2.2.2 Representation as a Tree

Given a binary tree in which each interior node stores a value, we can associate an implicit index with each value: the index for the value in node N is equal to the number of values stored to the left of the node (where "left" is defined using the standard prefix ordering). We refer to this as implicit indexing. See Figure 1 for an example, in which the indices are indicated beside the nodes.

The advantages of this representation is that retrieving the value for index i requires only log(i) steps, on average, if the tree is balanced, and insertion or removal of nodes automatically causes the appropriate adjustments to the indices of nodes to the right. The disadvantage is that the tree must be kept approximately balanced, which increases the costs of modifying the tree.

```
ImplicitlyIndexedTree = spec
type Core E =
  | Nil
  | Branch {value: E, left: IITree E, right:IITree E}
type IITree E = {core:Core E, size:Nat, height:Nat}
  | sizeOK?(core, size) && heightOK?(core, height)
```

To represent an implicitly indexed tree, we use a standard algebraic construction for the tree itself (with leaf nodes represented by Nil and interior nodes represented by Branch). We also cache the size of the tree since this is frequently referenced in indexing, and the height of the tree since this is frequently referenced in balancing. We add the semantic constraint that the cached size must be equal to the true size of the tree:

```
op [E] sizeOK?(core:Core E, size:Nat): Bool =
  case core of
  | Nil -> size = 0
  | Branch {value: E, left: IITree E, right:IITree E} -> size = 1 + left.size + right.size
```

Note that we use the cached values of the size for the sub-trees – i.e., only one node is checked by any single application of this constraint. An alternative would be to recurse through the tree and count the nodes afresh. The former is semantically equivalent to the latter since the former applies throughout the tree. However, the former is cheaper to compute and probably better suited to runtime monitoring using sampling.

The cached value of the height of the tree is likewise semantically constrained.

The exact structure of the tree reflects its history of node additions and removals, and concomitant rebalancing – the structure is not a function of the values stored in the nodes. Consequently, multiple trees may store the same contents at the same (implicit) indices; i.e., multiple trees may represent the same sequence.

We define two implicitly indexed trees to be equivalent iff they have the same values at the same

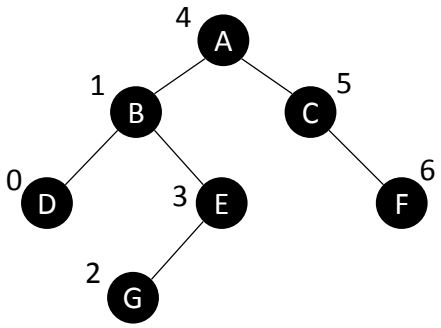

Figure 1 Implicit indexing in a binary tree

indices. The representation of a Finite Sequence is then the quotient type of Implicitly Indexed Tree over this equivalence relation.

3.2.2.3 Comparison of Representations

The list representation of a sequence is expected to be efficient for element insertion and removal, especially at the front of the sequence. The tree representation incurs (amortized) rebalancing costs for insertion and removal. However, the tree representation should be faster for random access to the elements, particularly for large sequences.

The list representation does not impose any semantic constraints. The tree representation imposes semantic constraints on the cached values of the tree's size and height which can be used in runtime monitoring.

3.2.3 Ordered Set

When we have a total order for some type, we can take advantage of the order in representing a set over the ordered type. For example, a set of integers can be represented by a sequence ordered according to the < relation (note that this implies that the list is non-repeating); a binary search can be used on the ordered list to determine if it contains some given value.

Ideally, we would be able to use an ordered set as an optimized representation of a standard set. However, this is not quite possible because additional information is required when an ordered set is initially constructed, namely the ordering relation. For example, the function to construct an empty, standard set takes no arguments, but the function to construct an empty, ordered set requires the ordering relation (even though the empty set contains no elements – the ordering must be available for subsequent element additions).

In some languages, a default order might be inferred from the element type. For example, a type class in Haskell might identify < as the default order for integers. However, even in such cases, the functions available on an ordered set exceed those available for a standard set. For example, an ordered set can yield its minimum and maximum elements, the i^{th} element, and ordered folds (a fold left or fold right, meaning the elements are processed according to the ordering relation or its inverse). Consequently, Ordered Set is treated as a distinct type in this work.

The core specification of Ordered Set is similar to the specification presented above for a standard set, except that there is an additional observer to retrieve the ordering relation from an Ordered Set, and operations such as adding an element are constrained to leave the order unchanged.

One problem arises because the order is observable: equality between two ordered sets requires them to have equal ordering relations. Since equality of functions (including ordering relations) is not executable, equality of ordered sets is not executable. (Though it is well defined as a specification construct.) An alternative function to test if two ordered sets contain the same elements is easily defined and is executable.

3.2.3.1 Representation as a Strictly Ordered List

An Ordered Set can be represented as a strictly ordered list:

StrictlyOrderedList = spec
type StrictlyOrderedList E = {order:StrictOrder E, data:List E} | strictlyOrdered?(data, order)

where a StrictOrder is a predicate p on E×E that is total ($x \neq y \Rightarrow p\ x\ y \vee p\ y\ x$), irreflexive (p x x

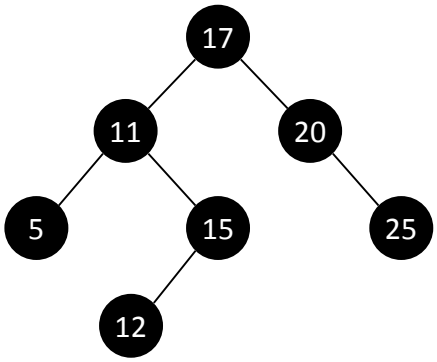

Figure 2 Strictly ordered binary tree

= false) and transitive ($p\ x\ y \wedge p\ y\ z \Rightarrow p\ x\ z$) – for example, the integer less-than predicate

Since there is exactly one strictly ordered list representing any given set, there is no need to form a quotient type (unlike for a list representation of a standard set).

3.2.3.2 Representation as a Strictly Ordered Finite Sequence

This representation is similar to the representation as a strictly ordered list, except that it uses a sequence as the core data structure. The difference between a list and a sequence is somewhat vague, but here we consider a list to offer sequential access to its members whereas a sequence offers random access.

3.2.3.3 Representation as a Strictly Ordered Binary Tree

An Ordered Set can be represented as a binary tree, in which values are stored in interior nodes and values to the left are strictly less than values to the right (where "left" and "right" are defined according to the standard prefix ordering). See Figure 2 for an example.

The advantage of this representation is that checking if a value is stored in the tree requires, on average, log(size of tree) steps if the tree is balanced. The disadvantage is that the tree must be kept approximately balanced, which increases the costs of modifying the tree.

The specification defines a standard algebraic tree structure in which leaf nodes are presented by Nil (and contain no data) and interior nodes are presented by Branch:

```
StrictlyOrderedBinaryTree = spec
type Core E =
  | Nil
  | Branch {value: E, left:OContents E, right:OContents E, min:E, max:E}
```

In addition to carrying a single value and its sub-trees, an interior node caches its minimum and maximum values, since these are frequently referenced by the tree operations.

Next, the specification defines a structure to carry the data that is common to interior and leaf nodes:

```
type Contents E = {core:Core E, order:StrictOrder E, size:Nat, height:Nat } |
                        ordered? && sizeOK? && heightOK? && minOK? && maxOK?
```

The primary constraint is that the values carried by the nodes be ordered:

```
op [E] ordered?(C:Contents E): Bool =
  case C.core of
  | Nil -> true
  | Branch {value: E, left:Contents E, right:Contents E, min:E, max:E}
    -> left ~= Nil => C.order(left.max, value) &&
       right ~= Nil => C.order(value, right.min)
```

This states that an interior node is ordered iff the value stored in the node is ordered with respect to the maximum value of its left sub-tree (if that is not a leaf node), and the minimum value of the right sub-tree (if that is not a leaf node). A leaf node is trivially well ordered.

The tree size, tree height, minimum value and maximum value are cached because these are frequently referenced. Each of the cached values is semantically constrained. For example, the cached size is constrained to be zero if the tree is a leaf, or with respect to the sizes of the sub-trees otherwise.

```
op [E] sizeOK?(C: Contents E): Bool =
  case C.core of
  | Nil -> size = 0
  | Branch {value: E, left:Contents E, right:Contents E, min:E, max:E}
-> struct.size = 1 + left.size + right.size
```

Note that the constraint references the sub-trees cached sizes – i.e., only one node is checked by any single application of this constraint. An alternative would be to recurse through the tree and count the nodes afresh. The former is semantically equivalent to the latter since the former applies throughout the tree. However, the former is cheaper to compute and probably better suited to runtime monitoring using sampling.

The cached height, minimum and maximum are likewise constrained.

The exact structure of a tree reflects its history of node additions and removals, and concomitant rebalancing – the structure is not a function of the values stored in the nodes. Consequently, multiple trees may store the same contents; i.e., multiple trees may represent the same ordered set.

We define two Strictly Ordered Binary Trees to be equivalent iff they store the same values. The representation of an Ordered Set is then the quotient type of Strictly Ordered Binary Trees over this equivalence relation.

3.2.3.4 Comparison of Representations

Many operations can be optimized under the constraint that the elements are ordered, regardless of which of the three representations are used: e.g., membership test, subset test, union and intersection.

If a sequence truly offers random access to its members, then binary search is an efficient way to test if a value is stored.

If a list truly offers only sequential access to its members, then binary search is not an efficient way to check if a value is stored, since the first comparison would need to traverse half the list. Rather, the test can terminate early once a member is found that does not satisfy the ordering relation with respect to the sought value.

The tree representation allows testing if a value is stored in a logarithmic number of steps.

All three representations involve some additional cost to maintain the invariants as the set is modified, compared with the list representation of a standard set. However, the rebalancing costs of the tree representation are likely to be highest.

All three representations provide semantic constraints that are well suited to sampling at run-time.

3.2.4 Finite Map

A finite map is an association between keys and values, with the keys in a given map being unique within that map. We thus characterize a finite map using two functions: to test if a particular key has an association in the map; and to retrieve the value associated with such a key.

```
FiniteMap = spec
type FiniteMap(K, V)
op [K, V] definedAt?(M:FiniteMap(K, V), key:K) infixl 20: Bool
op [K, V] @(M:FiniteMap(K, V), key:K | M definedAt? key) infixl 30: V

axiom map_equality is [K, V]
  fa(M1:FiniteMap(K, V), M2:FiniteMap(K, V))
    M1 = M2 <=>
    (fa(k:K) M1 definedAt? k <=> M2 definedAt? k) &&
    (fa(k:K) M1 definedAt? k => M1@k = M2@k)
```

3.2.4.1 Representation as an Association List

A straightforward representation of a finite map is as a list of key-value pairs, in which the keys are unique.

```
UnordAList = spec
op [K,V] uniqueKeys?(L:List (K*V)): Bool =
  fa (i:Nat, j:Nat) i<length(L) && j<length(L) => (L@i).1 = (L@j).1 <=> i = j

op [K,V] equiv?(L1:List (K*V), L2:List (K*V)): Bool =
  forall? (fn e -> e in? L2) L1 &&
  forall? (fn e -> e in? L1) L2

type UnordAList(K,V) = (List (K*V) | uniqueKeys?)/ equiv?
```

Since the key-value pairs are unordered, we form a quotient type over an equivalence relation that tests if two lists have the same contents regardless of order.

3.2.4.2 Representation as a Hash Map

An alternative representation partitions the key-value pairs into n buckets such that in bucket number i, each key k satisfies hash(k) mod n = i. This representation is similar to the hash set representation for finite sets (Section 3.2.1.2), so we will not discuss it further here.

3.2.4.3 Comparison of Representations

The pros and cons of the two representations are similar to those for the unordered list and hash set representations for finite set (see Section 3.2.1.3).

3.2.5 Ordered Map

If an ordering is available for the keys, the representation of a finite map can be optimized. The considerations are similar to those for an ordered set (see Section 3.2.3).

3.3 Generating All Implementations

As noted in Section 3.1.2, if App is a specification of some application that uses the abstract data type T, and if TasR is a morphism that shows how T can be represented using the concrete data type R, then App[TasR] is the refined specification that uses the concrete data type instead of the abstract data type. Such morphisms can be sequenced to refine multiple abstract data types – e.g., App[OrderedSetAsTreeM][FiniteSequenceAsListM]. The refinements are applied in order: i.e., first OrderedSet is refined, then FiniteSet.

When an application uses several abstract data types, the number of possible refinements is large – which of course is ideal for synthetic diversity. However, some refinements may introduce an abstract data type into an application, because the concrete type uses the new abstract type. Moreover, each abstract type must be refined if a fully executable version is to be produced. The order of refinements is also important: if abstract type T is introduced *after* T is refined into a concrete type, then the newly introduced references to T will *not* be refined.

The above considerations mean that it can be tedious to identify all refinements that are required and sequence them correctly for all implementations. Consequently, we implemented a simple tool to simplify the construction of all implementations:

- We manually list the abstract types *directly* used by the application.

- Knowledge about which abstract type uses which other abstract types is declaratively encoded in the tool.
- Knowledge about which abstract type refines to which representation is declaratively encoded in the tool.
- The tool generates all feasible refinement terms.

For example, finite map uses finite set. This is encoded as:

```
op FiniteMap:BranchPoint =
  mkBranchPoint("specs/Library/FiniteMap", "FiniteMap", [FiniteSet])
```

The use of the mkBranchPoint function indicates that there are (multiple) possible refinements for this type. The first argument is the path in the file system to the finite map specification. The second argument is a label for use in tracing the tools operation. The final argument is a list of other types that are needed by FiniteMap.

HashSet is fully executable (it needs no refinement) but it uses the finite sequence type:

```
op HashSet:BranchPoint =
  mkConcreteBranchPoint("specs/Library/HashSet", "HashSet", [FiniteSequence])
```

The use of the mkConcreteBranchPoint function is what indicates that this type does not require refinement.

The refinements for finite map are encoded as:

```
op FiniteMapAsHash:Refinement =
  mkRefinement(FiniteMap, [HashMap], ["specs/Library/FiniteMapAsHash#M"])
op FiniteMapAsList:Refinement =
  mkRefinement(FiniteMap, [UnorderedAList], ["specs/Library/FiniteMapAsList#M"])
```

The first argument to the mkRefinement function is the type to be refined. The second argument is a list of types into which it will be refined (in practice, there was always only one type). The final argument is a list of morphisms that will carry out the refinement.

The encodings are collected into libraries to simplify adding additional knowledge for new applications.

One way to think about the knowledge encoded is as a dependency graph, part of which is shown in Figure 3. Nodes in the graph are data types or algorithms. There are two types of arrows between nodes: one indicating refinement, the other indicating use. e.g., OrderedMap refines to StrictlyOrderedBinaryATree, but uses OrderedSet. (The tool requires that the graph be acyclic. It may be possible to relax this condition, in which case the tool would need to avoid revisiting any given node during the construction of any single implementation.)

Example output of the tool is shown in Table 2.

Although the discussion refers to types, the tool can also be used to refine algorithms.

The tool can also apply transformations (such as those that introduce run-time monitoring).

3.4 Measuring and Increasing Diversity

There are at least two situations in which we would like to have some measure on the degree of diversity in a set of implementations:

- When we are deploying one of the implementations at a time, but are continually or periodically changing which implementation – we would like to ensure that the next implementation to be deployed is sufficiently different from some number of preceding implementations.
- When we are deploying multiple implementations simultaneously, we would like to ensure that the deployed implementations are as different from each other as possible.

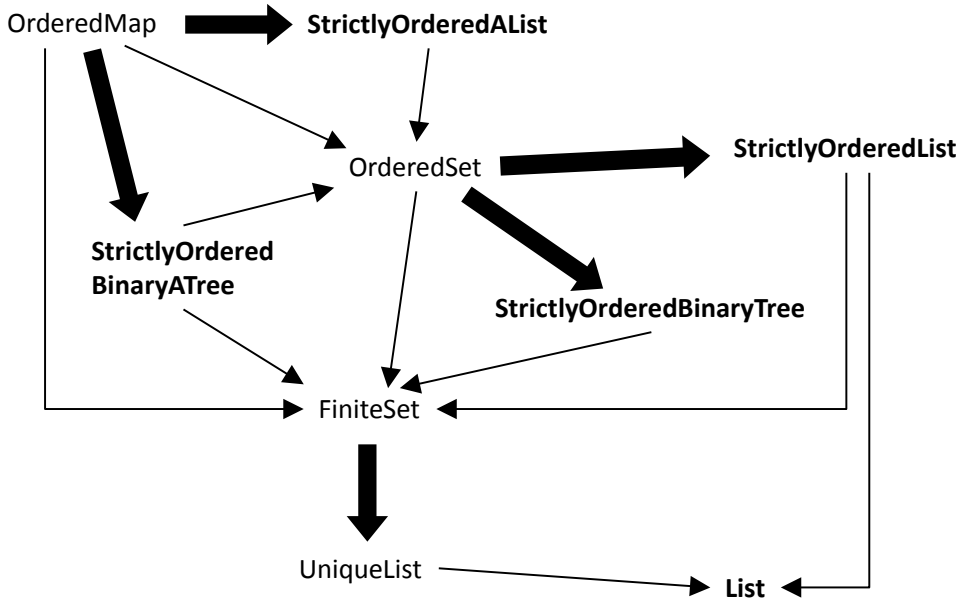

Figure 3 Dependencies between types

Thick arrows indicate that the source refines to the target.
Thin arrows indicate that the source type uses the target type.

Table 2 Output of diversity tool

version_0 = FiniteMapUnitTests#Tests
[../FiniteMapAsHash#M]
[../FiniteSetAsHash#M]
[../HashSet#configureForSeqAsList]
[../FiniteSequenceAsList#M]

version_1 = FiniteMapUnitTests#Tests
[../FiniteMapAsHash#M]
[../FiniteSetAsHash#M]
[../HashSet#configureForSeqAsTree]
[../FiniteSequenceAsTree#M]

version_2 = FiniteMapUnitTests#Tests
[../FiniteMapAsHash#M]
[../FiniteSetAsList#M]

version_3 = FiniteMapUnitTests#Tests
[../FiniteMapAsList#M]
[../FiniteSetAsHash#M]
[../HashSet#configureForSeqAsList]
[../FiniteSequenceAsList#M]

version_4 = FiniteMapUnitTests#Tests
[../FiniteMapAsList#M]
[../FiniteSetAsHash#M]
[../HashSet#configureForSeqAsTree]
[../FiniteSequenceAsTree#M]

version_5 = FiniteMapUnitTests#Tests
[../FiniteMapAsList#M]
[../FiniteSetAsList#M]

To these ends, we defined a correlation metric between two implementations of an application, as follows:

- For each pair of refinements, we defined a (symmetric) correlation matrix. For example, consider the three implementations for an ordered set discussed in Section 3.2.3: list, sequence and tree. The list and sequence implementations are more similar to each other than to the tree representation, so the correlation matrix for these might be as shown in Table 3. (Here, and below, it is convenient to refer to a refinement using the representation that it introduces, once the abstract data structure is known – e.g., List refers to the refinement OrderedSetAsList.)

Table 3 Correlation matrix for representations of ordered set

	List	Sequence	Tree
List	1.0	0.7	0.2
Sequence	0.7	1.0	0.2
Tree	0.2	0.2	1.0

- For two or more implementations, we define the correlation as a weighted sum of the pairwise correlations between their refinements. For example, Table 4 shows five *teams* of three implementations of a sorting function. Various sorting algorithms are shown in the column labeled "Sort". Various implementations are shown for three abstract data structures (finite sequence, finite set and ordered set). Each of the five teams has a low correlation, about 0.04. In contrast, any team in which all three implementations are the same would have a correlation of exactly 1.

3.4.1 Increasing Diversity

For small sizes, we can generate all possible teams of that size and measure their correlations. These measurements are plotted for team size 3 in Figure 4 for sorting and for a second function, clustering (see Section 4.5). The horizontal axis shows correlation and the vertical axis shows the percentage of teams that have equal or lesser correlation.

Table 4 Low-correlation teams of implementations of sorting

Team 1	Sort	FinSeq	FiniteSet	OrdSet
version 1	SortUsingOrderedSet	List	HashSet	StrictlyOrderedBinaryTree
version 2	RunsSort	List		
version 3	QuickSort	IndTree		
Correlation: 0.041666666666666664				

Team 2	Sort	FinSeq	FiniteSet	OrdSet
version 1	SortUsingOrderedSet	List	HashSet	StrictlyOrderedList
version 2	RunsSort	List		
version 3	QuickSort	IndTree		
Correlation: 0.041666666666666664				

Team 3	Sort	FinSeq	FiniteSet	OrdSet
version 1	SortUsingOrderedSet	List	HashSet	StrictlyOrderedSequence
version 2	RunsSort	List		
version 3	QuickSort	IndTree		
Correlation: 0.041666666666666664				

Team 4	Sort	FinSeq	FiniteSet	OrdSet
version 1	SortUsingOrderedSet	List	UnordList	StrictlyOrderedBinaryTree
version 2	RunsSort	List		
version 3	QuickSort	IndTree		
Correlation: 0.041666666666666664				

Team 5	Sort	FinSeq	FiniteSet	OrdSet
version 1	SortUsingOrderedSet	List	UnordList	StrictlyOrderedList
version 2	RunsSort	List		
version 3	QuickSort	IndTree		
Correlation: 0.041666666666666664				

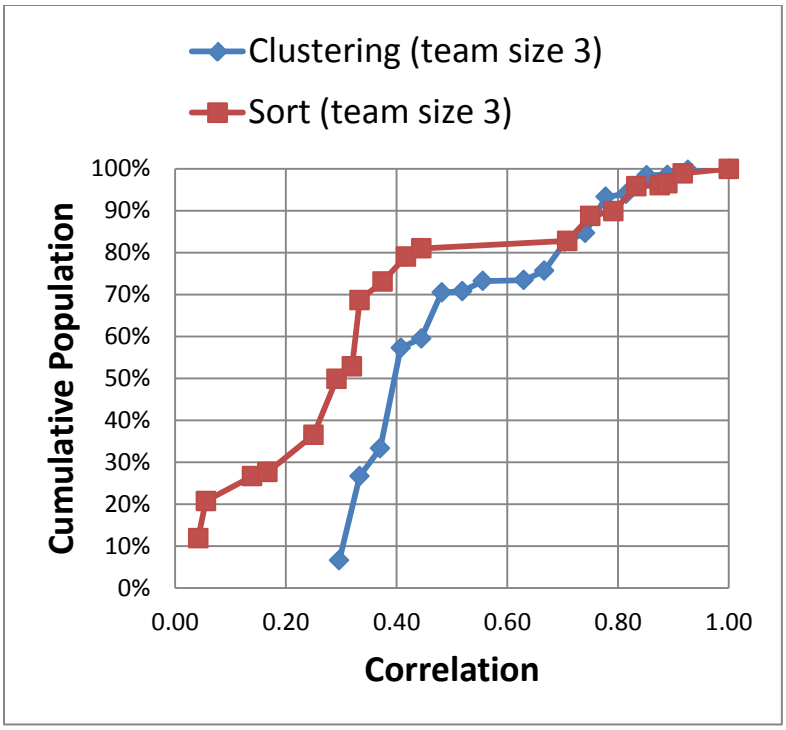

Figure 4 Distributions of correlations for sorting and clustering

The average correlation for sorting is 0.35 and for clustering it is 0.49. In comparison, the lowest correlations are 0.04 and 0.30. Picking teams at random would thus produce quite poor diversity. (This occurs in part because a particular algorithm may have many more possible implementations than its competitors simply because it uses more data types – random picks from among all possible implementations would thus be biased towards choosing this algorithm. In the weighting used to calculate the overall correlation of a team, algorithm choices are weighted more heavily than data representation choices.)

Consequently, we implemented a hill-climbing algorithm that, given an initial team, randomly modifies the team while keeping its size fixed, retaining the modification if it decreases the team's correlation. We found that this algorithm performed well in the test applications we tried. For example, its output is shown in Table 5 for sorting – starting with a randomly selected team, it rapidly approaches a minimally correlated team.

Table 5 Output of hill-climbing algorithm for teams of implementations of sorting

Team 1	Sort	FinSeq	FiniteSet	OrdSet
version 1	ExtractionSort	List		
version 2	ExtractionSort	IndTree		
version 3	SortUsingOrderedSet	IndTree	UnordList	StrictlyOrderedSequence
Correlation: 0.3194444444444445				

Team 2	Sort	FinSeq	FiniteSet	OrdSet
version 1	MergeSort	List		
version 2	ExtractionSort	List		
version 3	SortUsingOrderedSet	IndTree	UnordList	StrictlyOrderedSequence
Correlation: 0.055555555555555504				

Team 3	Sort	FinSeq	FiniteSet	OrdSet
version 1	QuickSort	IndTree		
version 2	MergeSort	List		
version 3	SortUsingOrderedSet	IndTree	UnordList	StrictlyOrderedSequence
Correlation: 0.041666666666666664				

This correlation metric seems well suited to algorithm and data representation diversity. It is not clear how to incorporate other forms of diversity, such as diversity generated from obfuscation techniques (see below).

3.5 Obfuscation Techniques for Diversity

Techniques from the field of code obfuscation can be adapted for synthetic diversity: more versions of the code for an application can be generated by applying different combinations of techniques at different strengths. The differences between implementations brought about by obfuscation techniques may not be as fundamental as with other diversification techniques, but they may nonetheless help to raise the costs for an attacker.

For this work, we developed a transformation to randomly reorder the arguments to functions. For example, given the following function

```
op solve(a:Int, b:Real, c:Bool): Real
```

reordering the arguments may generate the following:

```
op solve1(b:Real, c:Bool, a:Int): Real
```

Any call to the original solve function (including recursive calls) may be replaced with a call to the generated function, with actual arguments permuted appropriately. The likelihood of a function being reordered and the likelihood of a call being replaced are parameters of the transformation.

We also did preliminary development of a transformation to reorder record fields – however, this transformation is not as fully implemented. It should be straightforward to incorporate other obfuscation techniques such as the introduction of opaque predicates and code motion.

3.6 Run-time Monitoring & Repair

3.6.1 Generating Run-time Monitors

Specifications and refinements contain semantic constraints that can be used for run-time monitoring. The approach that we take here is to augment function entry and exit points with code that checks that a function's arguments and result satisfy their constraints.

For example, consider the specification below.

```
Div2 = spec
op div2(a:Nat, b:Nat, c:Nat | b + c ~= 0): Nat =
  a / (b + c)
end-spec
```

The div2 function has the constraint b + c ~= 0. We apply the addSemanticChecks transformation to create a new specification that incorporates run-time monitoring:

```
Mon = Div2{addSemanticChecks{checkArgs?: true, checkResult?: true, checkRefine?: true}}
```

The first two parameters to this transformation, checkArgs? and checkResult?, control the checking of function arguments and results against semantic constraints arising from types. The third argument, checkRefine?, controls the checking of function results against semantic constraints that arise from function refinements – see below.

The new specification contains a refined version of the div2 function, into which monitoring code has been inserted:

```
refine def div2 (a: Nat, b: Nat, c: Nat | b + c ~= 0): Nat
  = let (a, b, c) =
        SemanticError.checkPredicate(
          (a, b, c),
           fn (a0: Int, b0: Int, c0: Int) -> b0 + c0 ~= 0,
           fn _ -> "Subtype violation on arguments of div2")
    in
    a / (b + c)
```

The function SemanticError.checkPredicate performs the monitoring. The first argument is a value to be checked (a 3-tuple in this case). The second argument is the semantic constraint being checked: it is always a function so in this case it is expressed using an anonymous function. The third argument provides a label to be displayed if a violation is detected.

If no violation is detected, then the SemanticError.checkPredicate function returns the checked value unchanged, so the let binding has the same effect as: let (a, b, c) = (a, b, c); i.e., it is the identity. This structure is used in case the monitor detects a violation and returns a repaired value – see Section 3.6.2.

A similar construct is used for monitoring function results. For example, consider the function to return the absolute value of an integer:

```
op abs(a:Int): Nat =
  if a < 0 then -a else a
```

The Nat type is declared to be the non-negative integers. The transformation to add monitoring thus introduces a check that the result is non-negative:

```
refine def abs (a: Int): Nat
  = let result = if a < 0 then - a else a in
      let result =
        SemanticError.checkPredicate
          (result, fn (i: Int) -> i >= 0, fn _ -> "Subtype violation on result of abs")
      in result
```

Note that constraints can be stated explicitly, as in the div2 example, or be part of a type declaration, as in the abs example. In fact, Specware treats an explicit constraint as a local definition of an anonymous type.

A similar construction is used to check refinement constraints. For example, we can specify and give a definition for abs:

```
op abs(a:Int): Nat =
  the(b:Nat) b = a || b = -a
refine def abs(a) =
  if a < 0 then -a else a
```

The transformation introduces a check that the result of abs satisfies the refinement constraint as well as the type constraint:

```
refine def abs (a: Int): Nat
  = let result = if a < 0 then - a else a in
      let result =
        SemanticError.checkPredicate
          (result,
           fn (result0: Int) -> result0 = a || result0 = - a,
           fn _ -> "Result does not match spec for abs")
      in
      let result =
        SemanticError.checkPredicate
          (result, fn (i: Int) -> i >= 0, fn _ -> "Subtype violation on result of abs")
      in result
```

This example also shows how multiple constraints are checked, by sequencing calls to the monitor.

The transformation disregards any constraint that it determines to be non-executable; e.g., any constraint that, directly or indirectly, does unbounded quantification or that checks functions for equality.

The SemanticError.checkPredicate function is parameterized to allow it either to output a warning when it detects a semantic error, or to generate an error condition (similar to an

exception). An application may contain code to handle an error condition; if it does not, then the application will terminate.

The function is also parameterized to allow random sampling of the semantic constraints with a constant probability.

3.6.2 Generating Run-time Repair Mechanisms

The transformation that generates run-time monitoring code has an optional parameter for a repair operator that it applies if the semantic constraint is violated. (We refer to it as an operator rather than a function because it may randomly select a repair – thus it may not truly be a function of its inputs.)

For example, a possible repair operator for non-negative integers is:

```
fn i:Int -> if random(2) = 0 then –i else 0
```

This will randomly return the absolute value or 0. Which repair is the better depends on what we know about the types of disruptions that might affect the data.

3.6.3 Augmenting Data Structures to Enhance Monitoring & Repair

Some data representations are semantically unconstrained. For example, any pair of real numbers may be a valid complex number in the Cartesian representation. To enable run-time monitoring, such representations may be augmented with additional data that is derived from their original data: the augmented representation is then subject to a semantic constraint in the form of the relationship between the original and derived data.

For example, a field can be added that records a hash-code:

```
type CartesianPlusHash = {real:Real, imag:Real, h:Nat | h = Cartesian.hash(real, imag)}
```

Formally, the Cartesian and CartesianPlusHash types are isomorphic: they are in one-to-one correspondence. This allows a transformation to automatically replace all uses of the Cartesian type with the CartesianPlusHash type, with any needed computation of the hash fields automatically inserted.

The addSemanticChecks transformation can then be applied to introduce run-time monitoring based on the semantic constraint. Additional data such as this can also be used in repairing compromised data representations.

We developed a transformation to add a hash field to arbitrary structures and apply the isomorphic substitution.

In principle, any derived fields could be used to augment an unconstrained structure. For example, we might cache the value of the radius of a complex number:

```
type CartesianPlusRad = {real:Real, imag:Real, rad:Real | sqr(rad) = sqr(real) + sqr(imag)}
```

It should be possible to develop a transformation that, given a data representation and a list of observer functions, generates various augmentations of the representation, or even randomly constructed functions over the observers. However, we have not yet done this.

3.6.4 Using Multiple Representations Simultaneously

If there are multiple representations available for some abstract type, then a redundant representation can be constructed containing each of the singular representations. For example, a dual representation for a complex number contains both a Cartesian and a polar representation:

type ComplexDual = {c:Cartesian, p:Polar | c = p}

Such redundant representations are semantically constrained: the singular representations must all be (semantically) equal. The multiple representations can thus be used for cross-checking: if one representation is compromised, another may not be. A compromised representation can be repaired from the other, sound representations.

Note that equality between the representations is semantic rather than structural: a Cartesian representation and a polar representation both contain two real fields, but we cannot determine equality simply by comparing the fields of one with the fields of the other. A semantically sound equality test can be determined by looking for axioms of the form

value = constructor(observer1(value), observer2(value), …)

where each of the observers is defined on the abstract type rather than just on the representation. Such axioms tell us how to deconstruct and reconstruct a representation, and tell us that the set of observers is complete. Thus, any one set of observers could be used to test for equality.

For example, the Cartesian and polar specifications contain the following axioms:

c = mkCartesian(real(c), imag(c))
p = mkPolar(rad(p), arg(p))

The observers real & imag and the observers rad & arg are all defined on the complex type so we can test equality of complex numbers using either real & imag or rad & arg.

Moreover, to repair a compromised representation, we apply its constructor to values obtained by applying its observers to a sound representation. For example, to repair (i.e., reconstruct) a Cartesian representation c from a polar representation p:

c = mkCartesian(real(p), imag(p))

And vice-versa:

p = mkPolar(rad(c), arg(c))

When disagreement is detected among the representations, how do we determine which are compromised and which are sound? In general, voting schemes can be used. However, if each representation is semantically constrained, then we may assume that whichever representations violate their constraints are the ones that are compromised.

For example, a better dual representation for complex numbers uses the CartesianPlusHash representation instead of the Cartesian representation, since the former is semantically constrained. (The polar representation is semantically constrained.)

In addition, a computation on the abstract type can be implemented using each of the representations, and their results cross-checked. This may detect an error that exists in the code that is peculiar to one implementation. (Clearly it is best if the computations are as independent

as possible for each representation. This is not required in the approach to multiple representations discussed above, but it is possible.)

For example, the following function computes the complex roots of a quadratic equation:

```
op solve(a: Real, b: Real, c: Real | a~=zero): Complex * Complex =
  let discr: Complex = sqrt(sq b - 4 * a * c) in
  ( (-b + discr) / (2 * a), (-b - discr) / (2 * a) )
```

The sqrt function returns a complex value, so the addition, subtraction and division in the last line involve complex arithmetic. With the dual representation, the whole computation is performed using the Cartesian(PlusHash) representation, and again with the polar representation, and the results compared.

For this work, we developed a transformation that, given multiple representations for an abstract type, automatically constructs a redundant representation, generates code to cross-check and repair, if necessary, the representations, and generates code to perform redundant computations and cross-check the results.

3.6.5 Automatically Rotating Representations

A variation on the preceding is to automatically generate a 1-of-n representation from n representations of an abstract type: at any given time, only one of the representations is present, but if a computational error is detected, it is replaced with another representation and the computation retried. This may allow a computation to succeed if the error is present only in code peculiar to one representation.

For this work, we developed a transformation that, given multiple representations for an abstract type, automatically constructs a rotating representation.

4 RESULTS AND DISCUSSION

To evaluate our approach to synthetic diversity:

- We developed unit tests for each of the common data structures discussed in Section 3.2 and generated all implementations using combinations of representations. Each implementation passed the unit tests.
- We developed various sorting algorithms and generated all implementations of each algorithm using combinations of its supporting data structures. We showed that each implementation was functionally correct and measured its execution time on various classes of inputs.
- We developed a simple web application for solving quadratic equations, the roots of which may be complex. We introduced a sporadic error into the solver and showed that the run-time monitor could detect the error and switch representations.
- We developed a web application that stores spatial data (i.e., data that is associated with spatial coordinates) and can answer certain queries, such as returning all data points that lie within a given region. We used a data structure called a kd-tree to store the data – this structure has interesting semantic constraints. We added mechanisms to perturb the data and showed that the run-time monitor could detect compromises and (partially) repair the data.

- We specified a clustering function and developed two algorithms that use spatial queries. We compared their outputs.

These are discussed in detail below.

4.1 Unit Tests

For each of the common data structures discussed in Section 3.2, we developed unit tests derived from the semantic specifications of the data structures' functions, without reference to any implementation details. For example, the specification of the set union function is

```
op [E] \/(S1:FiniteSet E, S2:FiniteSet E) infixr 24: FiniteSet E =
  the(U) fa(e:E) e in? U <=> (e in? S1 || e in? S2)
```

For element e of the result, there are thus four cases:

- e is in S1 and in S2;
- e is in S1 but is not in S2;
- e is not in S1 but is in S2;
- e is not in S1 and not in S2.

To test this, we create four distinct lists:

- I, containing elements that are in both S1 and S2;
- D12, containing elements that are in S1 but not S2;
- D21, containing elements that are in S2 but not S1;
- C, containing elements that are in neither S1 nor S2.

We then construct S1 from the elements in I and D12, and we construct S2 from the elements in I and D21.

We then compute R = S1 \/ S2 and verify that:

- for each x in I, x is in R;
- for each x in D12, x is in R;
- for each x in D21, x is in R;
- for each x in C, x is not in R;
- for each x in R, x is in I, D12 or D21.

The underlying method is to use operations on lists (which we assume to be correctly implemented) to verify operations on sets.

Having developed the unit tests using just the abstract FiniteSet data structure, we generated implementations using the available representations for FiniteSet. We also generated versions with run-time monitors (Section 3.6.1). We performed the unit tests for each version – each test passed for each representation (after some errors were corrected).

Table 6 Number of implementations for unit testing

FiniteMap	12
FiniteSequence	4
FiniteSet	6
OrderedMap	40
OrderedSet	20

We did likewise for the other data structures. The number of versions for each data structure is shown in Table 6.

4.2 Sorting

Two algorithms may be functionally equivalent but exhibit different performance characteristics, and, in particular, different worst case behaviors. Using multiple algorithms leads to performance diversity, which makes it more difficult for an attacker to deliberately target an application with a denial of service attack.

As an illustration, we implemented several algorithms for sorting a (non-repeating) sequence. In addition to the standard QuickSort, MergeSort and BubbleSort algorithms, we implemented some less common algorithms:

- ExtractionSort: the elements of the input sequence are partitioned into those that are in order with respect to the next element (if any) and those that are out of order. If there are none that are out of order, the algorithm returns the original sequence; otherwise, it is recursively applied to the in-order and out-of-order partitions. We might expect this algorithm to be efficient if the input sequence is almost sorted.
- RunsSort: the elements of the input sequence are partitioned into subsequences, each of which is already sorted; these sorted subsequences are merged. We might expect this algorithm to be efficient if the input sequence is created by concatenating multiple sequences which are (almost) sorted.
- Since we have a data type for OrderedSet, we can sort a (non-repeating) sequence by creating an OrderedSet from it and then extracting the elements using OrderedSet's iterators.

We created (non-repeating) input sequences of various sizes and types – random, sorted, almost sorted (created by perturbing a sorted sequence), inverse sorted and almost inverse sorted – and timed the algorithms. The results are shown in Figure 5.

Aside from confirming that each diversified implementation did in fact correctly sort the input, the results show significant changes in relative performance among the algorithms for different types of input. For example, QuickSort is generally fast but not when the input is already sorted.

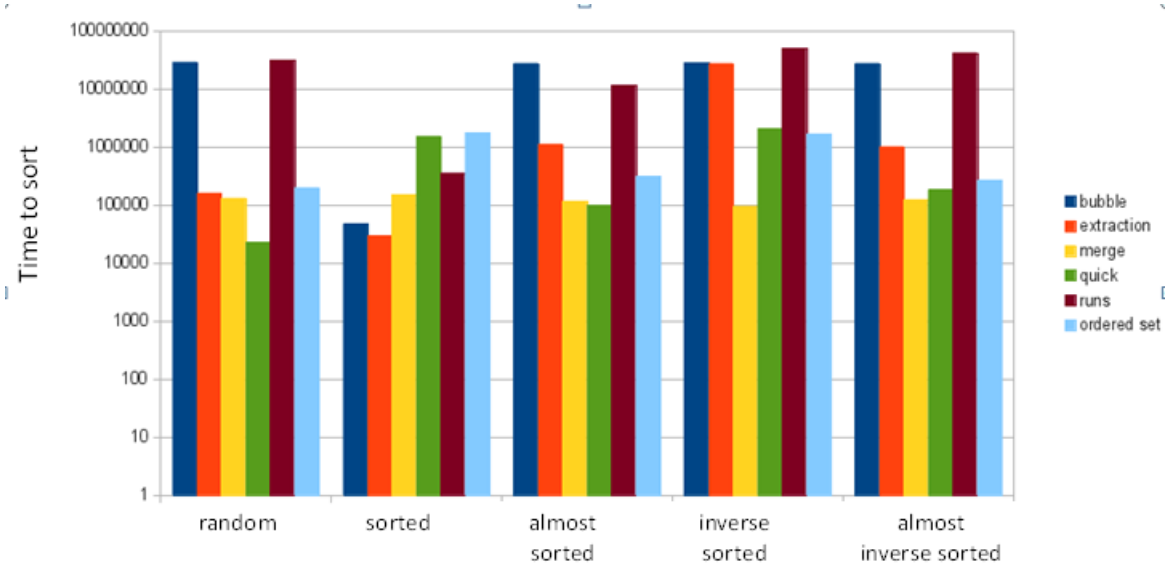

Figure 5 Performance characteristics of sorting algorithms

4.3 Quadratic Equation Solver

To demonstrate using dual and 1-of-n representations (Sections 3.6.4 and 3.6.5), we developed a web application that computes the roots to a quadratic equation. A quadratic equation has two roots given by $x = (-b \pm \sqrt{b^2 - 4ac}) / 2a$, where the roots may be complex, if the discriminant $b^2 - 4ac$ is negative. The computation of the roots may be expressed as follows:

```
op solve(a: Real, b: Real, c: Real | a~=zero): Complex * Complex =
  let discr: Complex = sqrt(sq b - 4 * a * c) in
  ( (-b + discr) / (2 * a), (-b - discr) / (2 * a) )
```

- For simplicity, the parameters of the equation are assumed to be real rather than complex.
- The first parameter, a, is constrained to be non-zero.
- The square root function (sqrt) returns a complex number. Thus, the subsequent addition, subtraction and divisions also involve complex arithmetic.

The application uses a client-server architecture (see Figure 6):

- the server contains generated code that computes the roots of quadratic equations;
- the client sends quadratic equations to the server and displays the roots, along with any log messages from the server.

For expediency, the server code is embedded in a free web server, called Hunchentoot [10], and the client is implemented in JavaScript/HTML. The generated code is associated with a URL on the web server, with the parameters of a quadratic equation passed using standard HTTP URL notation (e.g., http://127.0.0.1/solve?a=1.1&b=7.2&c=-3).

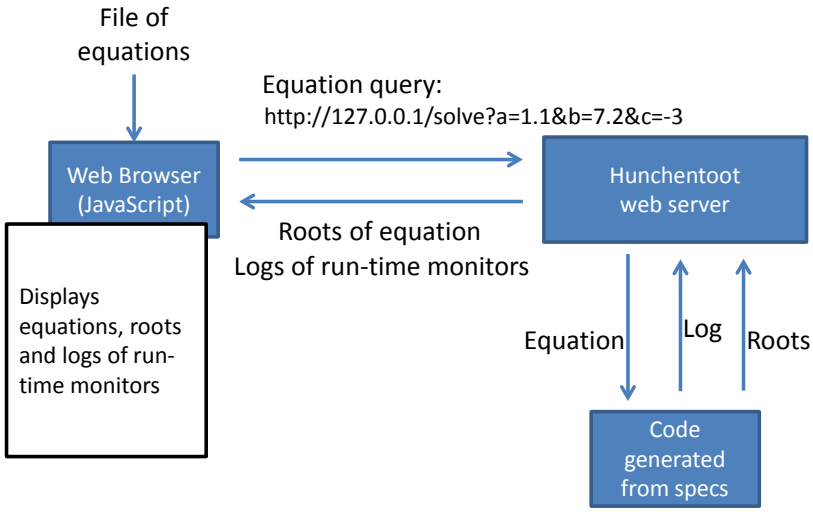

Figure 6 Client-server architecture for quadratic equation solver

As discussed above, we can use two representations of complex numbers that are semantically constrained, and thus allow for run-time checking:

- a Cartesian representation in which the x & y coordinates are augmented with a hash-code: CartesianPlusHash = (real, imag, h | h = hash(real, imag));
- a Polar representation in which the radius is constrained to be non-negative, the angle is constrained to the range $[0, 2\pi)$, and zero is uniquely represented (if the radius is 0, then the angle must be 0).

For the demonstration, we modified the code that computes the roots so that, at random, it may change the discriminant to violate these invariants. When the run-time monitor detects the violations, it retries the computations either with the same representation of complex numbers or with the alternative representation. These actions are logged and reported to the client for display.

Figure 7 shows the server starting with a Cartesian(PlusHash) representation, solving one equation without error, but then encountering an error on the second equation and switching to the polar representation. On the ninth equation, another error is encountered and the representation switches back to Cartesian.

4.4 Spatial Queries

We developed a web application for a demonstration of run-time monitoring using a kd-tree. This is a spatial decomposition data structure for sets of data points associated with spatial coordinates. The data structure is designed to support fast queries of the forms: which data points lie within a given region, or which data point is closest to given coordinates?

The web application allows:

- a set of points to be generated in 2-dimensional space;
- a region of the space to be selected, and the points within that space found;
- the underlying kd-Tree data structure to be attacked in various ways;

Equation				Cartesian				Polar			
#	a	b	c	Root		Root		Root		Root	
1	3.1	47.3	203.4	-7.63	2.72	-7.63	-2.72				
changing cartesian discr Failed with restartCount 0 Trying other representation...											
2	7.5	-121.7	611.0					9.03	1.42°	9.03	18.31°
3	-2.9	54.1	-304.3					10.24	18.40°	10.24	1.34°
4	4.8	-21.1	30.8					2.53	1.63°	2.53	18.10°
5	-1.4	26.5	-126.7					9.51	19.42°	9.51	0.32°
6	6.4	-29.2	97.9					3.91	2.98°	3.91	16.76°
7	1.3	24.8	158.1					11.03	8.22°	11.03	11.52°
8	-3.9	-39.0	-277.8					8.44	12.81°	8.44	6.93°
changing polar discr Failed with restartCount 1 Trying other representation...											
9	8.9	65.5	543.6	-3.68	6.89	-3.68	-6.89				
10	6.7	97.6	355.9	-7.28	0.26	-7.28	-0.26				
11	-4.2	51.8	-165.5	6.17	-1.17	6.17	1.17				

Figure 7 Quadratic equation solver switching representations

- the error reports and repairs on the kd-Tree data structure, produced by the run-time monitors, to be examined.

The user interface runs on a web browser, and communicates with code that we generate for the kd-Tree data structure (and run-time monitors) which is integrated with the Hunchentoot web server – see Figure 8. The communication is structured using the JSON (JavaScript Object Notation) standard, which provides conventions for light-weight serializing and deserializing of structured data; modern web browsers provide JSON functionality; a free plug-in provides JSON functionality on the server.

One of the main components of the user interface is the map display. This shows the locations of the data points and allows the user to select points by clicking and dragging-out a rectangle. We

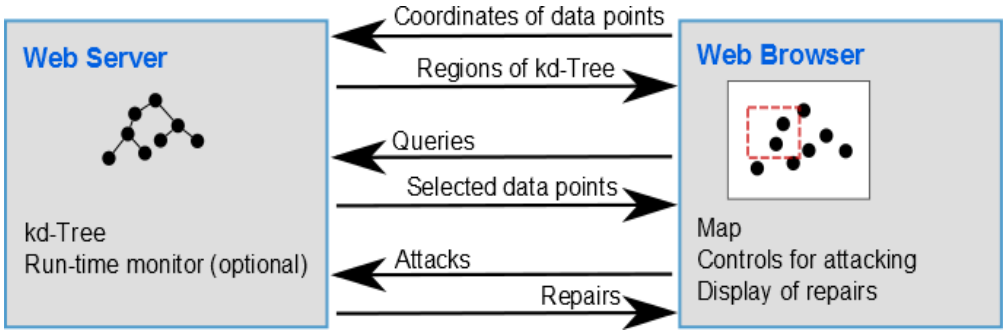

Figure 8 Architecture of kd-tree demonstration

implemented the map using an SVG object (*Scalable Vector Graphics*). With SVG, we add geometric shapes and text declaratively, and the SVG object takes care of rendering their representations on the HTML page – when a data point is selected, we merely change its *style* and the SVG object efficiently updates the rendering.

The remaining components of the interface (panels to display the coordinates of the selected points and controls for attacking the kd-Tree) were straightforward to implement using standard HTML and CSS. The logic of the interface was implemented in JavaScript.

An important aspect of the user interface is the visualization of the errors detected and repairs made by the run-time monitor – these are presented in an intuitive manner.

Details of the kd-Tree data structure and the demonstration are given in Appendix A.

4.5 Clustering

Clustering is a technique for automated data classification: a set of data points is partitioned into multiple clusters such that the points within a single cluster are similar (in some sense) and points in different clusters are dissimilar. We implemented two clustering algorithms:

- Centroid clustering, using the k-Means algorithm [12]: each data point is to belong to the cluster with the nearest mean;
- Density clustering, using the DBSCAN algorithm [13]: clusters are formed so that no point in a cluster is more than a certain distance from the nearest neighbor in the cluster.

The k-Means algorithm requires the number of clusters (k) to be given in advance, so we complement it with a *silhouette* algorithm [14] that tries to find an optimal k.

4.5.1 Centroid Clustering: k-Means & Silhouette

The k-means algorithm takes a set of data points and outputs k partitions, where k is an input parameter to the algorithm. k-Means can be thought of as operating on a set of k cluster centers: each data point is associated with the nearest center. k-Means operates as follows:

1. A set of k points is generated as an initial guess at the cluster centers.
2. Each point in the data set is assigned to the cluster center that is closest (with random tie breaking if it is equidistant to two or more centers).
3. The set of points assigned to each cluster center is taken to be a partition. The mean of the coordinates of each partition is calculated, giving k new centers. If these new centers are sufficiently close to the previous centers, the algorithm terminates with the new centers as its output; otherwise, the algorithm repeats from Step 2 with the new cluster centers. (In the event that a cluster center has no assigned points, the algorithm may restart with a new set of centers.)

k-Means only guarantees (approximate) local optimality – for a different set of initial centers, it may output a different set of results.

Step 2 of the algorithm can be optimized using a data structure such as a kd-Tree (see Appendix A). The basic insight is that geometric reasoning can sometimes be used to determine that a particular center could not be the closest center for any point contained within an entire branch of the tree. Thus, Step 2 can be performed as follows:

i. Start at the top of the tree with the complete set of centers.

ii. If the number of centers has been reduced to 1, then assign every data point within this part of the tree to the sole remaining center. This is the main source of the algorithm's efficiency.

iii. Otherwise:

   a. If we are at a leaf node, iterate through all centers for each data point to find the closest center.

   b. If we are at a branch node, move into each child with only those centers that might be the closest center for some point contained in that child. Repeat from Step ii.

The filtering of the centers in Step iii.b uses the "blacklisting" algorithm [15]. This involves a dominance relation between centers, with respect to a given space *S*: center *c* dominates center *d* if-and-only-if *c* is closer than *d* to every point in S (where "point" here refers to the set of geometrical points in the space rather than the data points). For a given sub-tree, the centers are compared pair-wise and if one is dominated by any other, with respect to the space assigned to the sub-tree, then it can be eliminated from the set of centers. (The spaces are assigned to the sub-trees during construction of the kd-Tree, and are recorded in the kd-Tree.)

4.5.1.1 Silhouette

To try to automatically determine a good k for k-Means, we implemented a wrapper algorithm based on the silhouette metric. Given a set of clusters of data points, the silhouette of a point measures how well that point fits within its assigned cluster, versus the other clusters. The average of the silhouettes over the data points is then a measure of how well the data have been clustered. Given some lower and upper bounds on the number of clusters, the k-Means wrapper iterates through the range and finds some number of clusters with the best metric.

Per the Wikipedia definition, the silhouette *s(i)* of data point *i* is defined as:

$$s(i) = [b(i) - a(i)] / \max\{a(i), b(i)\}$$

where:

- a(i) is the dissimilarity of point i with respect to i's assigned cluster;
- b(i) is the lowest of the dissimilarities of point i with respect to the other clusters.

The dissimilarity of a point with respect to a cluster is the average of the distances between that point and each (different) point in the cluster. Note that for k-Means, a cluster contains those data points for which the cluster's centroid is closer than any other cluster's centroid. The computation of the silhouette can thus be optimized by replacing the average distance between a point and the points in a cluster with the distance between that point and the cluster's centroid.

If a point i has been assigned to a suitable cluster, then b(i) in the above term will be large compared with a(i) – the point will be closer to the other points in its assigned cluster than to the points in the other clusters. Thus s(i) will approach +1. Contrariwise, if the point has been badly assigned, then s(i) will approach -1.

4.5.2 Density Clustering

Given a set of data points, the DBSCAN algorithm picks a point (e.g., at random), determines if there are enough neighbors, and if so, starts a cluster containing that data point.

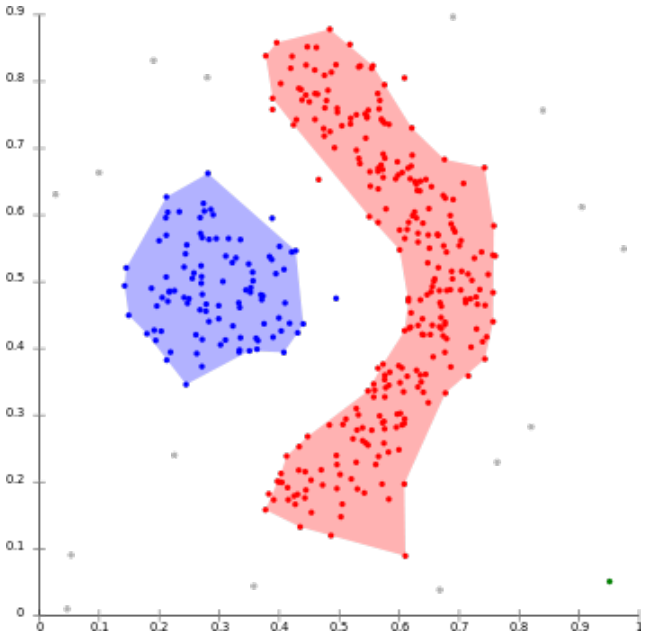

Figure 9 Clusters formed by DBSCAN (source WikiPedia)

It then considers each of those neighbors as a candidate for addition to the cluster: if the neighbor itself has enough neighbors, it is added to the cluster. The cluster is thus expanded by adding additional data points that are neighbors of points already in the cluster, so long as those points have enough neighbors.

Once this process completes for a given cluster, it is repeated with the data points that have not yet been considered for inclusion in any cluster.

Points with too few neighbors are not put into any cluster – they are considered noise.

The clusters formed by DBSCAN may be irregular in shape. DBSCAN may thus be better suited to some data sets than k-Means. For example, Figure 9 shows two clusters (plus noise data points) formed by DBSCAN.

4.5.3 Demonstration

We demonstrated the use of the two clustering algorithms for large-grained diversity; i.e., we developed a single specification for a clustering demonstration and then generated different implementations by diversifying over k-Means (with Silhouette) and DBSCAN, and over the lower level libraries on which these are based (in particular, the spatial decomposition library and common data structures).

In total, 48 implementations were generated from the single specification. Each implementation was run on the same set of data.

- Each implementation of k-Means produced the same clusters. (This required making k-Means deterministic by providing an algorithm for selecting the initial set of *k* points it uses to form the initial clusters – often, these initial points are selected randomly.)
- Each implementation of DBSCAN produced the same clusters.

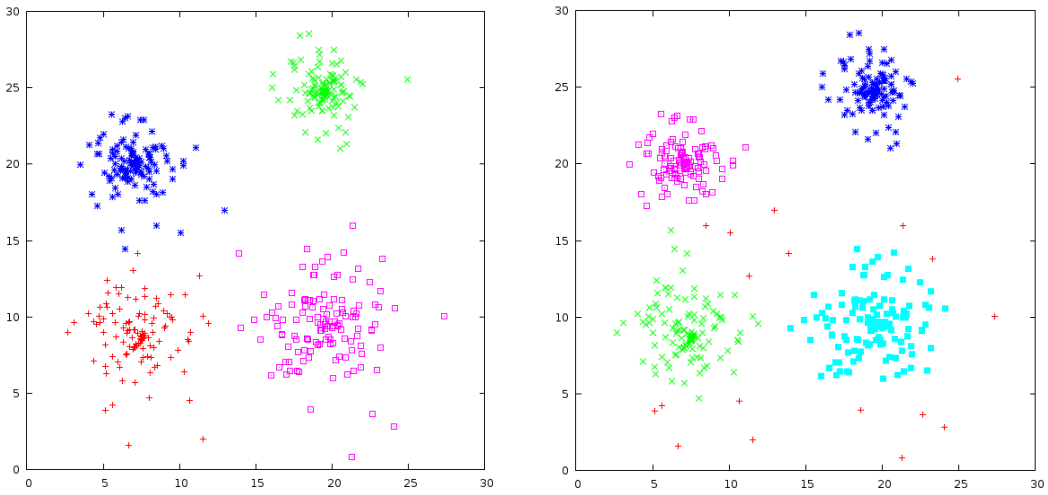

Figure 10 Examples of clusters produced by k-Means (left) and DBSCAN (right)

- The clusters produced by k-Means were not exactly the same as those produced by DBSCAN, but they are similar. For example, Figure 10 shows clusters produced by k-Means and DBSCAN (note that the latter leaves some points unassigned to any cluster).

For this case, the data set was generated using a mixture of 2-dimensional Gaussians and so both k-Means and DBSCAN are well-suited for clustering the data. In other cases, k-Means (in particular) or DBSCAN may not be appropriate for the data set, and may produce poor clusters.

Diversification over these two algorithms is thus qualitatively different from diversification over data type representations, for which each variant is required to produce exactly the same output. Clustering may be considered a "heuristic" problem, in that even though we can precisely specify either centroid clustering or density clustering, it may not be clear which version of clustering we should use for a particular application. We cannot even directly compare the results produced by the two algorithms since they use different metrics – determination of which produces better clusters would have to come from a higher level component that uses the clusters.

Nevertheless, there are still circumstances in which it is useful to diversify over these two algorithms.

- If we know the data comes from a mixture of Gaussians, then either algorithm should produce good clusters. They can thus be used as other types of diversification are used – to cross-check each other's results (allowing some tolerance for small differences in the clusters) and to reduce an adversary's ability to learn how an application works.
- k-Means may be significantly faster than DBSCAN (since it can use spatial reasoning to assign a group of data points to a cluster in a single step), particularly if the number of clusters can be well estimated, so that the silhouette-based wrapper for k-Means that tries to find the optimal number of clusters has a small range to iterate over. However, it will produce poor clusters if given non-Gaussian data, in which case DBSCAN might be used as a fall-back.

5 CONCLUSIONS

We investigated the use of semantics-rich specifications and refinements in medium-grained synthetic diversity. We showed how an abstract specification style combined with formal refinement can be used to generate large numbers of implementations of an application. We showed how semantic information in the specifications and refinements can be used to generate run-time monitors that check for and partially repair compromised data.

We developed several demonstration applications: a simple quadratic equation solver, sorting, a spatial query server and a clustering algorithm. These illustrate our techniques and technologies on moderately complex applications.

6 REFERENCES

1. Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. *Intrusion detection: Randomized instruction set emulation to disrupt binary code injection attacks*. 10th ACM Conference on Computer and Communication Security (CCS), pp 281 – 289. October 2003.
2. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. *Countering Code-Injection Attacks With Instruction-Set Randomization*. 10th ACM International Conference on Computer and Communications Security (CCS). October 2003.
3. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (CCS '04). ACM, New York, NY, USA, 298-307.
4. Adrienne Porter Felt, Kate Greenwood, and David Wagner. *The effectiveness of application permissions*. Proceedings of the 2nd USENIX conference on Web application development. USENIX Association, 2011.
5. Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. *Secure Execution via Program Shepherding*. USENIX Security Symposium. Vol. 92. 2002.
6. A. Sabelfeld and A. C. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications, 2003.
7. Specware – http://www.specware.org/
8. Isabelle/HOL proof system – http://www21.in.tum.de/~nipkow/LNCS2283/
9. Coq proof system – http://coq.inria.fr/
10. Hunchentoot web server – http://weitz.de/hunchentoot/
11. kd-Tree data structure – http://en.wikipedia.org/wiki/K-d_tree
12. k-Means clustering algorithm – http://en.wikipedia.org/wiki/K-means_clustering
13. DBSCAN clustering algorithm – http://en.wikipedia.org/wiki/DBSCAN
14. Silhouette clustering metric – http://en.wikipedia.org/wiki/Silhouette_(clustering)
15. Dan Pelleg and Andrew Moore: *Accelerating exact k-means algorithms with geometric reasoning*. Technical Report CMU-CS-00105, Carnegie Mellon University, Pittsburgh, PA.

Appendix A RUN-TIME MONITORS FOR KD-TREES

Abstract: This demonstration shows the automatic generation of run-time monitors from semantic constraints. It is based around a simple web application that shows points on a 2-dimensional map and allows the user to click-and-drag a rectangular region on the map to select points. As the user clicks-and-drags, the web front-end continually sends queries to a server to determine which points lie within the selected region. The server uses a spatial decomposition data structure to efficiently determine which points are selected. Specifically, the server uses a kd-tree, which recursively partitions a k-dimensional space one dimension at a time, until the number of points contained in a partition is sufficiently small. (In this demonstration, k is 2.) When determining which points lie in a given region, the kd-tree uses the recursive partitioning to restrict the query to only those partitions that intersect the region – for a typical query, these partitions contain only a small fraction of the points in the whole space.

The kd-tree data structure has various strong semantic constraints: e.g., when a space is partitioned, the resulting "children" must exactly cover the space. These semantic constraints can be used to generate monitors that check the constraints at run-time – if a constraint fails, the monitor initiates repair actions. To show the monitors in action, the web front-end has an "attacker" mode that allows the kd-tree data structure to be corrupted, and a "repair" mode that shows the violations found by the monitors and the repairs they produced.

A.1 Spatial Decomposition using kd-Trees

A spatial decomposition tree recursively splits a large space that contains many data points into a set of sub-spaces that each contains a subset of the points, namely those points that lie in the sub-space. There are several ways to organize the partitioning. In the particular case of kd-trees, each split involves 1 of the k dimensions; successive splits may involve different dimensions. Again, there are several ways to choose which dimension to split, and where the split should occur. In this demonstration, the largest dimension is split in half.

For example, Figure 11 shows 19 points in a space that is partitioned 5 times.

To determine which points lie within some given query region, a simple recursive algorithm is used that, at each node in the tree, checks if the region intersects the sub-space covered by the node; if not, then the node cannot contain any points that lie in the region and the algorithm does not proceed any further into this part of the tree – it returns the empty set (of points). Contrariwise, if the sub-space does intersect the region, then the branch may contain points that lie in the region, so the algorithm proceeds into both of its children and unions their result sets.

Eventually, the algorithm reaches a leaf node that contains no child trees. A leaf node contains at most n points, where n is some fixed number (here, 5). The algorithm then iterates through the points to determine which lie within the query region.

For typical queries, most of the tree can be quickly pruned off, requiring an in-depth search of only a small fraction of the tree, and consequently only a small fraction of the points must be individually checked against the region.

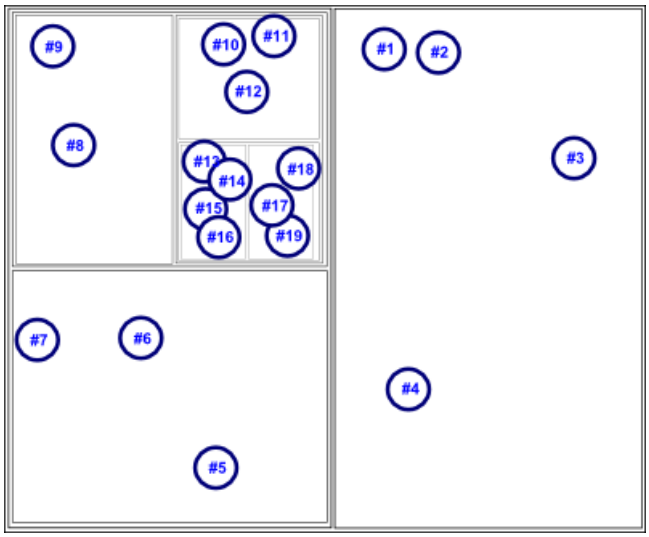

Figure 11 Example of spatial decomposition

Example of spatial decomposition – each circle represents a data point, each rectangle represents a node of the kd-tree

A.1.1 Semantic Constraints on kd-Trees

Typically, the nodes in a kd-tree contain minimal information, to optimize the use of memory. However, for this demonstration, each node contains additional information to support the run-time detection and repair of compromised nodes. Specifically, each non-leaf node contains the following information:

- the space covered by the node (recorded as a pair of k-dimensional points which form "opposite" corners of the space);
- the dimension in which the space is split;
- the coordinate at which the space is split. (In principle, this need not be recorded as it is the mid-point of the dimension. However, recording it allows other splitting schemes, such as splitting at the median coordinate of the data points.)

Each leaf node contains the following information:

- the space covered by the leaf node;
- the "id" of each point contained in the leaf;
- the points themselves.

Based on this information, the following constraints apply:

- The "lower" (respectively, "upper") child of a node has a space that is exactly the lower (resp., upper) part of the node's space, with respect to the recorded splitting dimension and coordinate.
- The set of points contained in a leaf node must have ids that exactly match the set of ids recorded in the leaf node.

- If a point is contained in a leaf node, the coordinates of the point must lie within the leaf node's recorded space.

A.1.2 Attacks on kd-Trees

This demonstration is based on a scenario in which the attacker has gained unauthorized access to the contents of the leaf nodes of the kd-tree but not to the interior nodes. This might be because the two are stored separately – many queries would need fast access to the interior nodes but not to the contents of the leaves, since most of the leaves do not need to be examined in a typical query. Thus, the interior nodes might be stored in main memory but the leaves might be stored off-line, lazily loaded from disc or a database as needed – they may even be stored on a separate computer.

For example, Figure 12 shows the structure of the kd-tree for the preceding space and points (Figure 11). Node A is the root and covers the space (0, 0) to (60, 50). Its "upper" child is B, which covers the space (30, 0) to (60, 50); B is a leaf node containing points #1 - #4. The "lower" child of A is the interior node C, which covers the space (0, 0) to (30, 50), etc.

Table 7 shows some of the leaf nodes – the spaces that they cover (in terms of a pair of coordinates), and the points that they contain, along with the coordinates of those points

Table 7 Leaf nodes

Leaf	Lower Coordinates	Upper Coordinates	Points
B	(30, 0)	(60, 50)	#1 (33, 47) #2 (37, 44) #3 (55, 35) #4 (35, 15)
D	(0, 0)	(30, 25)	#5 (17, 5) #6 (13, 17) #7 (3, 16)
F	(0, 25)	(30, 50)	#8 (5, 37) #9 (3, 46)
…			

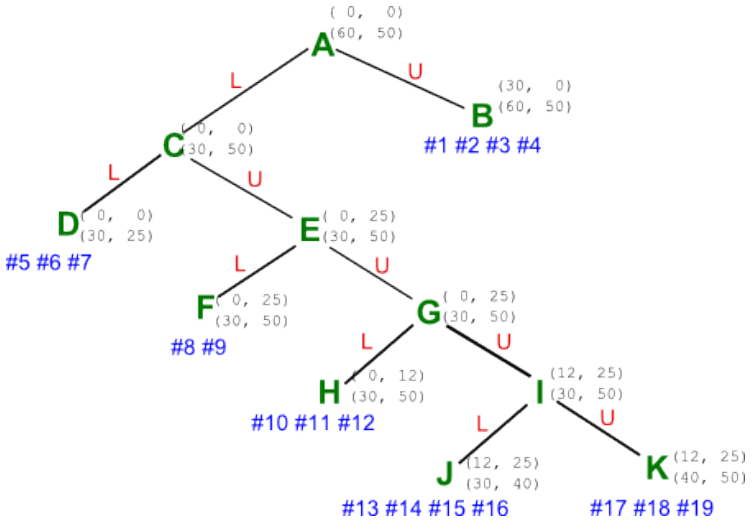

Figure 12 Example of kd-tree

In line with the above discussion, 4 types of attack are supported in this demonstration:

- The coordinates of a leaf can be changed. Doing so allows an attacker to effectively "hide" the points contained in the leaf, since the search algorithm will prune off the leaf based on its coordinates.
- The coordinates of the points stored in a leaf can be changed.
- Points can be deleted from a leaf.
- Spurious points can be added to a leaf.

(For simplicity, it is assumed that the ids of points are not changed, and any point added by the attacker has an id that is distinct from those of the existing points.)

A.1.3 Repairs Effected by the Run-time Monitors

The run-time monitors that are generated from the semantic information discussed above are able to detect the following problems and effect the following repairs:

- If a child node's space is not correct with respect to its parent's space, the recorded splitting dimension and the recorded splitting coordinate, then the child node's space is reset to a proper partition of its parent's space.
- If the coordinates of a point have been altered to such an extent that they fall outside of the space of the leaf node that contains the point, then the point's coordinates are reset to a random position within the leaf node's space.
- When the set of ids of the points currently contained in a leaf node is compared with the recorded set of ids the leaf node is supposed to contain, if there are missing points then they are regenerated with random coordinates within the leaf node's space.
- Likewise, if there are extra points, then they are deleted from the leaf node.

If extra information were recorded, it might be possible to effect more accurate repairs:

- If the points' data is stored redundantly, then it might be fully recovered.
- If the mean of the points' positions were recorded, then the random generation of coordinates might be tuned to match the recorded mean. Likewise for higher moments.

A.2 Walkthrough

The following sections detail how the demonstration works, and show examples of the user interface.

A.2.1 Generate some Points

When you load the demonstration, the web front end generates some data points in the x-y plane and sends them to the server. The server constructs the kd-tree and, for the purposes of this demonstration, sends the coordinates of each node (interior and leaf) back to the browser. (In a real application, the browser would likely not need the node coordinates.) The browser displays the points as discs on the map (Figure 13).

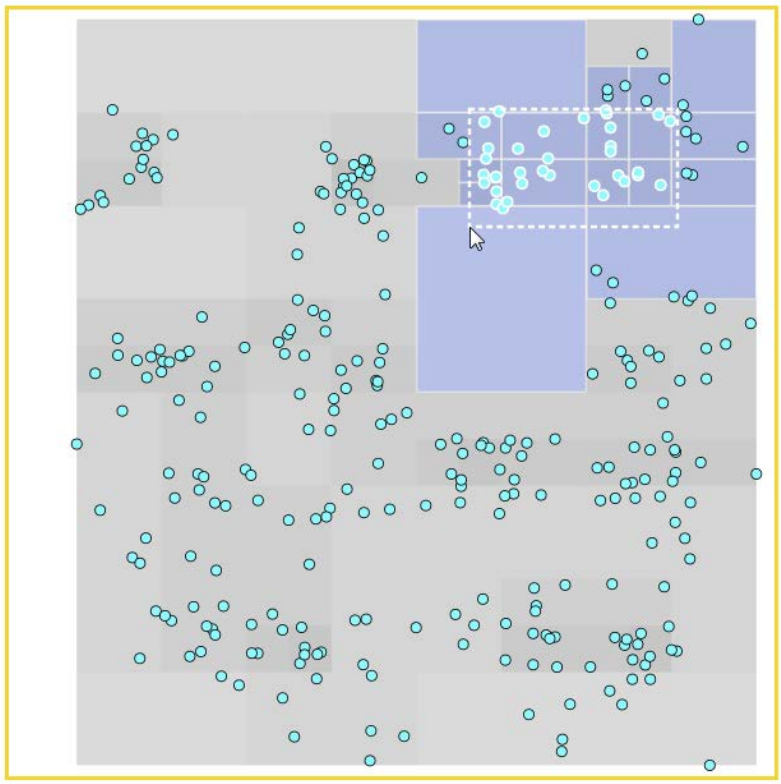

Figure 13 Display of data points, with some selected

The browser also displays the nodes as semi-transparent rectangles – since a parent node overlaps its ancestors on the plane, the deeper the tree, the darker the combined shading of the rectangles representing the nodes. In turn, the depth of the tree reflects the density of points.

If you refresh the web page, a new set of points is generated.

A.2.2 Make a Selection

Click and drag on the map to draw out a selection region.

As you drag, the control panel on the right shows the coordinates corresponding with the region, and the browser continually sends the coordinates to the server. The server uses the kd-tree to efficiently find the points that lie within the selection region and sends them back to the browser, which displays them (Figure 14).

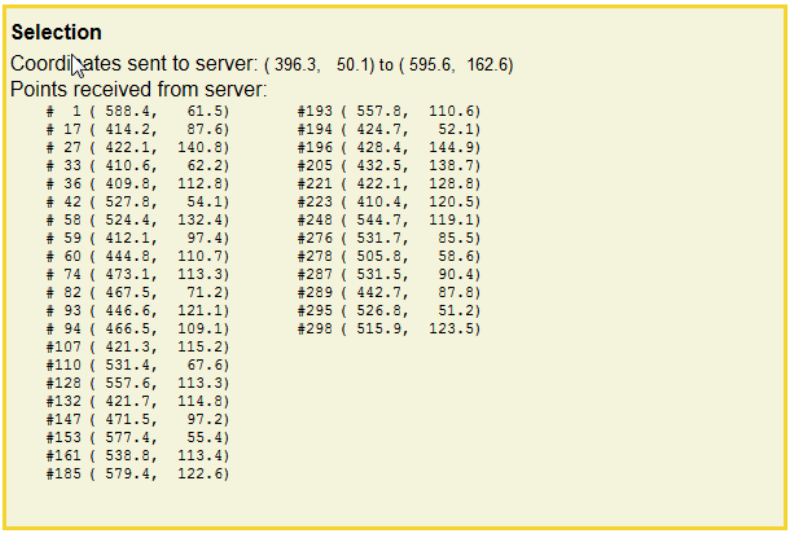

Figure 14 Display of the selected data points

A.2.3 Become the Attacker

Using the radio buttons near the top of the control panel, select the attacker role.

For the attacker role, the control panel displays a table of the leaf nodes that tile the x-y plane (Figure 15).

A.2.4 Edit a Tile

Select a tile by clicking on a row in the table, or by clicking on the map.

Select one that contains several data points.

Click the "edit" link on the row or double click to bring up the tile editor (Figure 16).

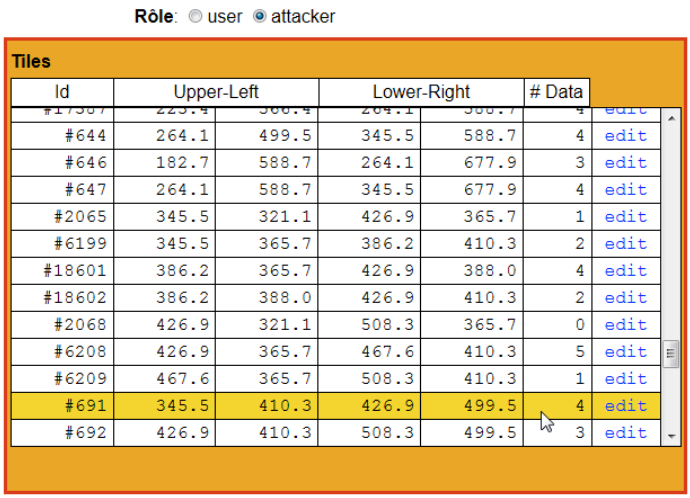

Figure 15 Display of the leaf nodes

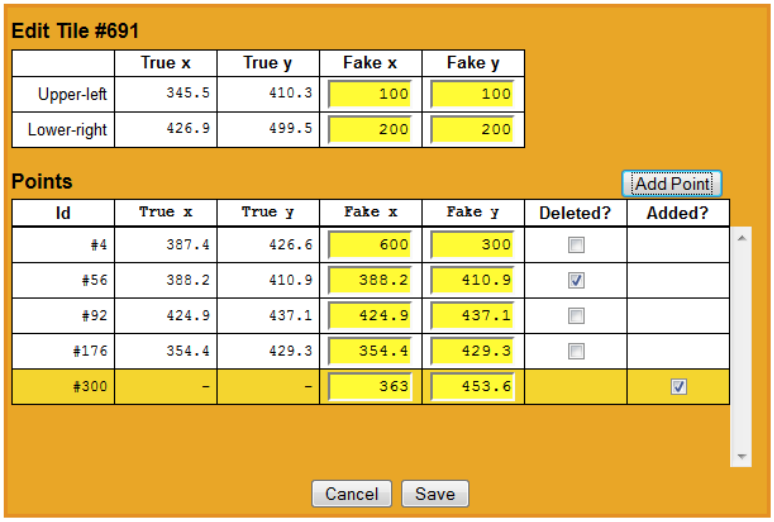

Figure 16 The tile editor

The upper part of the editor displays the coordinates of the tile. The lower part displays the points contained in the tile along with their coordinates.

Use the text boxes in the upper part of the editor (labeled "Fake x" and "Fake y"), enter new coordinates for the tile.

(The x-coordinate increases left-to-right on the map; the y-coordinate increases top-to-bottom on the map. As you move from one text box to the next (using the tab key or mouse), the tile's new location is displayed on the map. When you try to save the edit, the upper-left x-coordinate cannot be greater than the lower-right x-coordinate; likewise for the y-coordinate.)

Save the edit, select another tile/row and open the editor again.

Using the text boxes in the lower part of the editor, change the coordinates of one of the points.

As you change the coordinates, the true and fake positions are displayed on the map (in green and red, respectively). Choose a new position that lies outside the tile.

Remove a point by clicking one of the check boxes labeled "Deleted?".

The point is now displayed in black on the map. When the edit is saved, the point will be removed from the server's data. However, the web application will remember the point and display it in the tile editor. Subsequently unchecking the "Deleted?" box will cause the point to be reinserted back into the server's data.

Insert a fake point by clicking the "Add Point" button.

This causes a new point to be inserted into the tile, with random coordinates. It is displayed in fuchsia on the map. The point can be subsequently removed by unchecking the "Added?" box.

Save the edit.

The map will display the edits (Figure 17) because the attacker role is still selected.

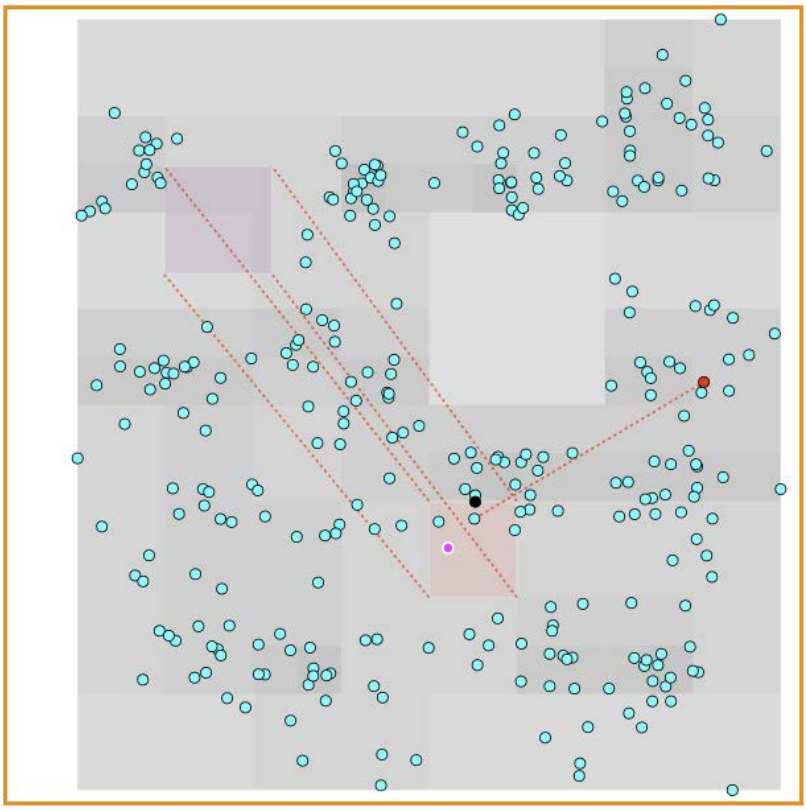

Figure 17 Map displaying the edits

A.2.5 Become the User Again

Using the radio buttons near the top of the control panel, select the user role.

The map no longer displays the edits to the tiles, since the user would normally have no knowledge of them.

Note that the point that was removed is no longer present, the point that was added is present and can be selected, and the point that was moved is displayed at its fake coordinates and will be selected if the selection region contains the fake coordinates and intersects the tile containing the point. You can toggle between the attacker and user roles to compare.

(An alternative manifestation of the attack would be that the moved and deleted points continue to be shown at their original coordinates, but cannot be selected. It depends on whether we consider the user's display to be refreshed before or after the attack takes place.)

Drag out a selection region on the map covering the tile whose coordinates were edited.

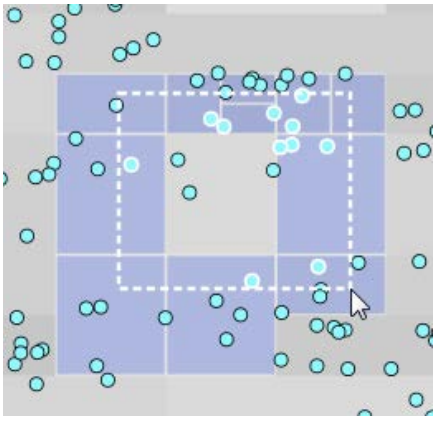

Figure 18 A masked sub-tree

Note that the points in the tile cannot be selected as normal (Figure 18). In order to select them, you would need to drag out a region that contains both the true and fake coordinates of the tile. If the fake coordinates are far off the map, say (1000000, 1000000) x (1000000, 1000000), then the points are effectively hidden from selection.

A.2.6 Activate the Run-time Monitors

When the demonstration is loaded, the server loads a version of the code that does not contain run-time monitors (so that you can verify the problems caused by the tile edits in the preceding section).

Using the radio button at the top of the control panel, select the version of the code that has the run-time monitors (Figure 19).

This causes the server to load the monitor-enabled version of the code. In this demonstration, the data uses the same representation for both versions of the code, and so it does not need to be translated when the code version is changed.

Drag out a selection region that intersects one of the edited tiles.

Doing so causes the server to receive the selection region's coordinates and start processing the kd-tree, looking for nodes that intersect that region. As it processes the tree, the automatically generated monitors check the nodes and, when a node is encountered that violates the kd-tree's semantic constraints, the monitors initiate repairs to the tree. For this interactive demonstration, the entire tree is checked and repaired where necessary so that all of the repairs can be presented together; an alternative strategy is to repair only the node whose constraint violation was initially encountered.

The map is updated to reflect the repairs (Figure 20)

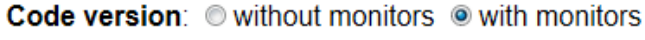

Figure 19 Selection of the version of code with run-time monitors

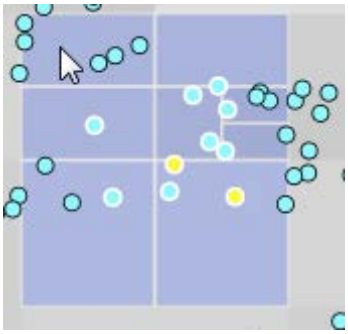

Figure 20 Repairs shown on the map

A.2.7 Examine the Repairs

The repairs are also displayed in a control panel (Figure 21).

Select each repair in turn by clicking on its rows in the control panel.

Each repair is displayed on the map:

- Figure 22 – the run-time monitor detects a violation of the constraint that a tile's coordinates be a partitioning of its parent's coordinates. The compromised coordinates are shown in red. The monitor makes an exact repair (with respect to the parent's coordinates) – the restored coordinates are shown in green.
- Figure 23 – the run-time monitor detects a violation of the constraint that a point's coordinates lie within the coordinates of the tile in which the point is stored. The compromised coordinates are shown in red. The monitor makes an approximate repair by generating random coordinates within the point's tile. The restored coordinates are shown in

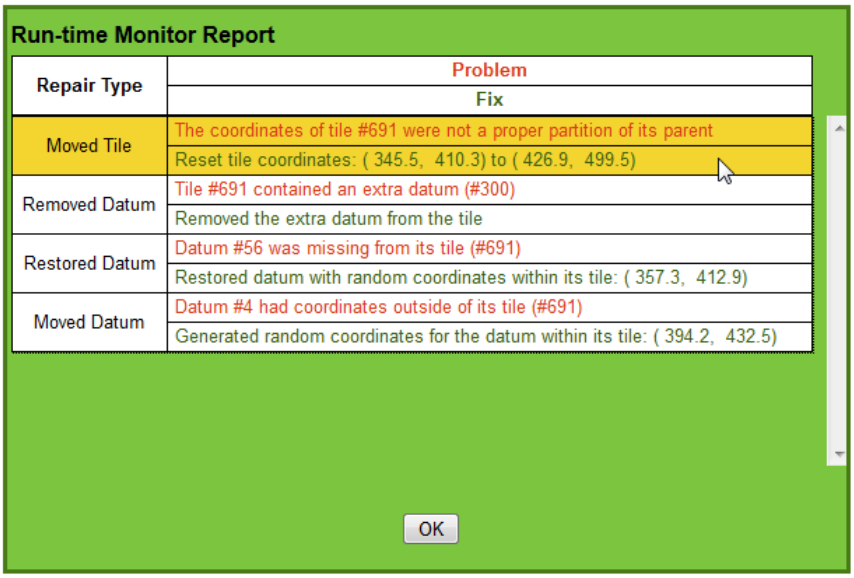

Figure 21 Control panel for repairs

yellow.

- Figure 24 – the run-time monitor detects that the red point is stored in a tile that does not have a record of the point's id, in violation of a semantic constraint. The monitor removes the point.
- Figure 25 – the run-time monitor detects that a tile has a record of an id that does not belong to any of the points stored in the tile, in violation of a semantic constraint. The monitor generates a point with the id at random coordinates (shown in yellow) within the tile's coordinates. This is an approximate repair.

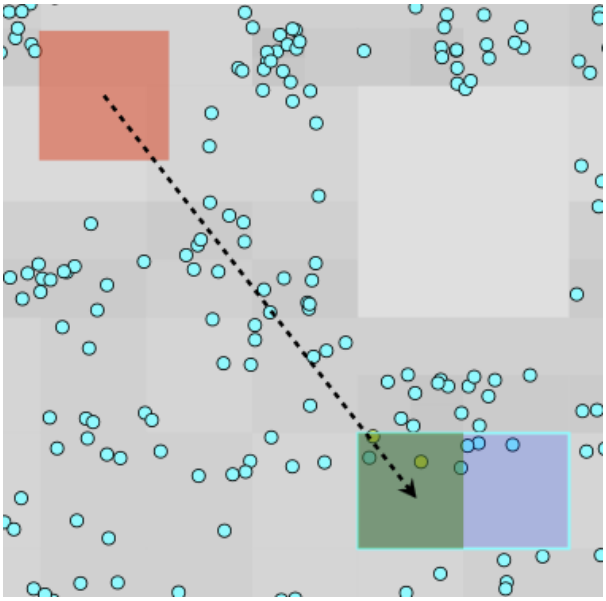

Figure 22 Tile coordinates restored

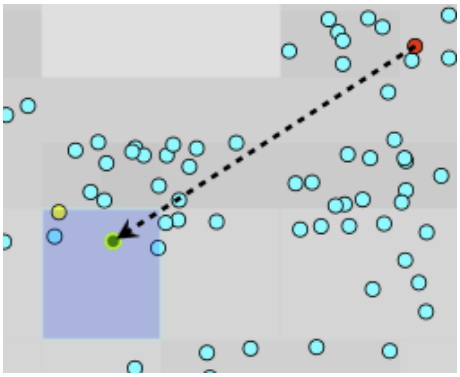

Figure 23 Point restored to a approximate coordinates

Figure 24 Spurious point removed

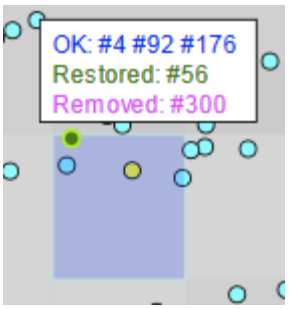

Figure 25 Deleted point restored

List of Symbols, Abbreviations and Acronyms

CPU	central processing unit
CSS	cascading style sheets
DBSCAN	density-based spatial clustering of applications with noise
HTTP	hypertext transfer protocol
HTML	hypertext markup language
JSON	JavaScript object notation
SQL	structured query language
SVG	scalable vector graphics
URL	uniform resource locato